

## Lecture #26: Fast Fourier Transforms – I

Professor: S. Lennart Johnsson

TA: Wei Ding

# 1 The Fast Fourier Transform

The Fast Fourier Transform, the FFT, is one of the most widely used algorithms in science and engineering. It is used both in signal processing and for the solution of partial differential equations. For instance, FFTs are used in so-called Fast Poisson solvers. It is also used in combination with cyclic reduction for the FACR (Fourier Analysis Cyclic Reduction) [4] method for Poisson's equation. FFTs are also used for fast convolution.

We will

- derive the Cooley–Tukey [1] FFT in both *decimation-in-frequency* and *decimation-in-time* form.
- carry out a detailed review of twiddle factor use. Which data use what twiddle factor is particularly important for distributed FFT with precomputed twiddle factors.
- present two different approaches to parallel FFT. In the first approach pairs of processors share in the computation of each butterfly computation, the fundamental computation in the FFT. In the other approach, the processors divide up the set of butterfly computations with a butterfly computation being performed entirely within a node.
- present only so-called *in-place* FFT. Such algorithms require no temporary arrays. However, the result appear in a “scrambled” order known as bit-reversed order with respect to the input.
- discuss reordering algorithms for the two types of parallel FFT.

## 1.1 Cooley–Tukey FFT

The Discrete Fourier Transform (DFT) is defined by

$$X(l) = \sum_{j=0}^{P-1} \omega_P^{lj} x(j), \quad \forall l \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}.$$

and the Inverse Discrete Fourier Transform (IDFT) is defined by

$$x(j) = \frac{1}{P} \sum_{l=0}^{P-1} \omega_P^{-lj} X(l), \quad \forall j \in [0, P-1], \quad \omega_P = e^{-\frac{2\pi i}{P}}.$$

Both the Forward and the Inverse Discrete Fourier Transforms constitute a matrix–vector multiplication with a special matrix. The matrix entries, or coefficients  $w_P^{lj}$ , are known as *twiddle factors*. The scaling factor  $P$  can be applied as above to one of the transforms, either the forward or the inverse, or by  $\sqrt{P}$  to both the forward and the inverse transform. In the matrix–vector form the DFT is written

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ \vdots \\ X(P-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_P^1 & \omega_P^2 & \omega_P^3 & \dots & \omega_P^{P-1} \\ 1 & \omega_P^2 & \omega_P^4 & \omega_P^6 & \dots & \omega_P^{2(P-1)} \\ 1 & \omega_P^3 & \omega_P^6 & \omega_P^9 & \dots & \omega_P^{3(P-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_P^{P-1} & \omega_P^{(P-1)2} & \omega_P^{(P-1)3} & \dots & \omega_P^{(P-1)(P-1)} \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(P-1) \end{pmatrix}$$

or

$$X = Wx$$

The IDFT formally is

$$x = W^{-1}X$$

The expression above for the IDFT have the form

$$\begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(P-1) \end{pmatrix} = \frac{1}{P} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_P^{-1} & \omega_P^{-2} & \omega_P^{-3} & \dots & \omega_P^{-(P-1)} \\ 1 & \omega_P^{-2} & \omega_P^{-4} & \omega_P^{-6} & \dots & \omega_P^{-2(P-1)} \\ 1 & \omega_P^{-3} & \omega_P^{-6} & \omega_P^{-9} & \dots & \omega_P^{-3(P-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_P^{-(P-1)} & \omega_P^{-(P-1)2} & \omega_P^{-(P-1)3} & \dots & \omega_P^{-(P-1)(P-1)} \end{pmatrix} \begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ \vdots \\ X(P-1) \end{pmatrix} = \frac{1}{P} W'X$$

It remains to be shown that indeed  $\frac{1}{P}WW' = I$ , where  $I$  is the identity matrix. Consider element  $(i, j)$  of the matrix product  $WW'$ . It is

$$\sum_{k=0}^{P-1} \omega_P^{ik} \omega_P^{-kj} = \sum_{k=0}^{P-1} \omega_P^{(i-j)k}$$

If  $i = j$ , then element  $(i, i) = P$ . If  $i \neq j$ , then we have

$$\sum_{k=0}^{P-1} \omega_P^{(i-j)k} = \omega_P^{(i-j)P} - 1 = 0$$

$P$	$2P(P - 1)$	$5P \log_2 P$
$2^{10}$	$2^{21} - 2^{11}$	$50 \cdot 2^{10}$
$2^{20}$	$2^{41} - 2^{21}$	$100 \cdot 2^{20}$
$2^{30}$	$2^{61} - 2^{31}$	$150 \cdot 2^{30}$

Table 1: Arithmetic operations for DFT evaluated as matrix–vector product and FFT.

Thus, we have shown that in fact  $\frac{1}{P}WW' = I$ . The IDFT can be computed just as the DFT, except the conjugated values of the twiddle factors are used.

The Cooley–Tukey [1] Fast Fourier Transform is an efficient way of computing the matrix–vector product, such that instead of requiring  $2P(P - 1)$  arithmetic operations  $5P \log_2 P + O(P)$  operations are required. Table 1 offers a comparison between the operation count for the two methods.

The *decimation-in-frequency* FFT is derived as follows. The idea is to arrive at a recursion formula such that the DFT on  $P$  data elements is expressed in terms of DFTs on data sets of size  $P/2$ . Thus,

$$X(l) = \sum_{j=0}^{\frac{P}{2}-1} \omega_P^{lj} x(j) + \sum_{j=0}^{\frac{P}{2}-1} \omega_P^{l(j+\frac{P}{2})} x(j + \frac{P}{2}) = \sum_{j=0}^{\frac{P}{2}-1} \omega_P^{lj} (x(j) + (-1)^{l'} x(j + \frac{P}{2}))$$

where we have used the facts that  $\omega_P^{\frac{P}{2}} = -1$ . Now, consider even and odd  $l$  separately. Then,

$$X(2l') = \sum_{j=0}^{\frac{P}{2}-1} \omega_P^{2l'j} (x(j) + x(j + \frac{P}{2})) = \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{l'j} (x(j) + x(j + \frac{P}{2})), \quad \forall l' \in [0, \frac{P}{2} - 1].$$

Similarly,

$$X(2l' + 1) = \sum_{j=0}^{\frac{P}{2}-1} \omega_P^{(2l'+1)j} (x(j) - x(j + \frac{P}{2})) = \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{l'j} \omega_P^j (x(j) - x(j + \frac{P}{2})), \quad \forall l' \in [0, \frac{P}{2} - 1],$$

where we used the fact that  $\omega_P^{2l'} = \omega_{\frac{P}{2}}^{l'}$ . We have now expressed the DFT as two separate DFT each on half as many data points and each computing half as many components. Thus, we have completed the derivation of a recursion step.

For the first recursion step the required computations are

$$(x(j) + x(j + \frac{P}{2})), \quad \forall j \in [0, \frac{P}{2} - 1]$$

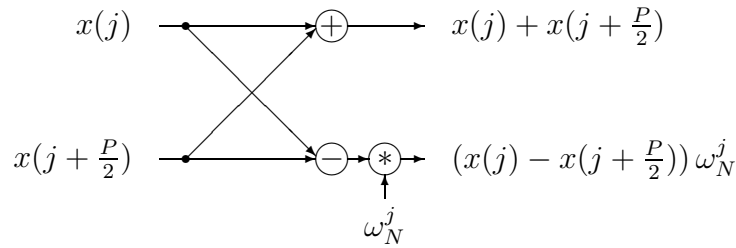


Figure 1: A butterfly.

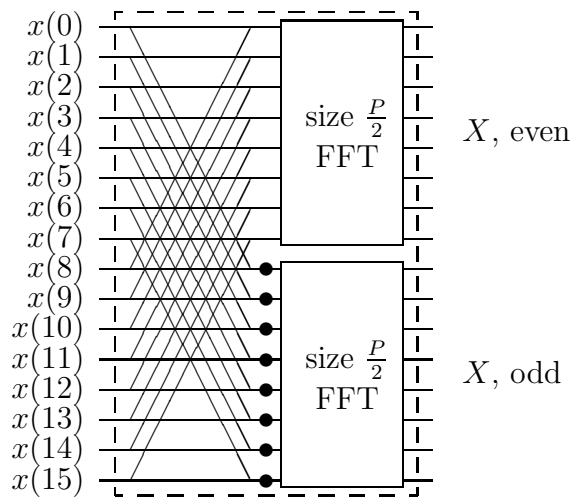


Figure 2: Decomposition of the discrete Fourier Transform.

and

$$\omega_P^j(x(j) - x(j + \frac{P}{2})), \quad \forall j \in [0, \frac{P}{2} - 1].$$

These computations, illustrated in Figure 1, are known as butterfly computations.

The first recursion step is illustrated in Figure 2. It shows the decomposition into even and odd components of the output through the butterfly computations. The “•”s represent complex multiplication by the twiddle factors.

The formulas

$$X(2l') = \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{l'j} (x(j) + x(j + \frac{P}{2})), \quad \forall l' \in [0, \frac{P}{2} - 1]$$

$$X(2l' + 1) = \sum_{j=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{l'j} \omega_P^j (x(j) - x(j + \frac{P}{2})), \quad \forall l' \in [0, \frac{P}{2} - 1],$$

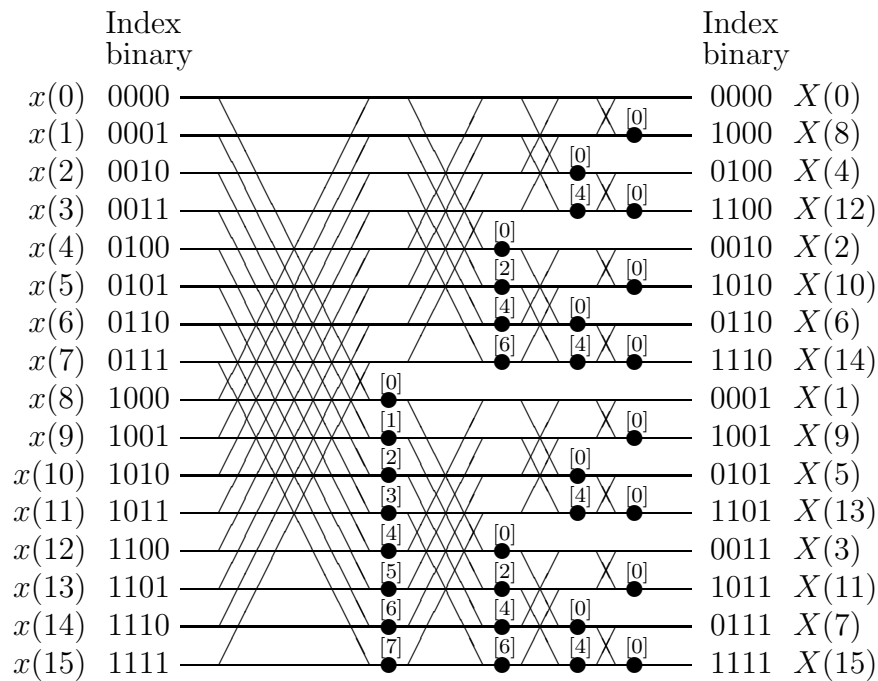


Figure 3: A 16-point decimation-in-frequency FFT.

are known as the *decimation-in-frequency* (DIF) splitting formulas. Applying them recursively  $\log_2 P$  times yields a decimation-in-frequency FFT for a data set of  $2P$  points. The Cooley–Tukey FFT is limited to transforms on sequences of lengths being powers of two. Other transforms, such as the prime factor transform, allows for more general sequence lengths.

The result for a 16 point FFT is shown in Figure 3.

The DIF FFT in Figure 3 is an *in-place* FFT. The result of each butterfly computation is written back to the same locations from which the input data was retrieved. No temporary array storage is required in an in-place algorithm.

Note that the ordering of the output data is different from that of the input data. The output order is obtained from the input order by reversing the binary code for the corresponding input index. Thus, for instance (0001) reversed becomes (1000), (0010) becomes (0100), etc. This order is known as *bit-reversed* order. Figure 2 shows addresses and their bit-reversed counterparts for four-bit addresses.

**Theorem 1** *A bit-reverse permutation is its own inverse.*

The bit-reversed output order is developed recursively by sorting even results before odd results. For example, in the first splitting formula all odd results will appear before all even results in

Index	Binary code	Reversed binary code	Bit-reversed index
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Table 2: Binary addresses and their bit-reversed values.

order from top to bottom, with each butterfly computation being made in-place. This ordering in fact implies that the least significant index bit has assumed the role of the most significant address bit for the result. The next application of the splitting formula has the same effect on the next bit of the result, i.e., within the even results all even results with respect to the least significant bit within its group (the second least significant bit) are ordered before any odd result within the group of even results, etc.

The *decimation-in-time* (DIT) Cooley-Tukey FFT [1] is derived by considering odd and even data points:

$$X(l) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_P^{l(2j')} x(2j') + \sum_{j'=0}^{\frac{P}{2}-1} \omega_P^{l(2j'+1)} x(2j'+1) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{lj'} x(2j') + \omega_P^l \sum_{j'=0}^{\frac{P}{2}} \omega_{\frac{P}{2}}^{lj'} x(2j'+1).$$

Now, consider the first and second halves of the transformed components.

$$X(l) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{lj'} x(2j') + \omega_P^l \sum_{j'=0}^{\frac{P}{2}} \omega_{\frac{P}{2}}^{lj'} x(2j'+1), \quad \forall l \in [0, \frac{P}{2} - 1],$$

and

$$X(l + \frac{P}{2}) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{lj'} x(2j') - \omega_P^l \sum_{j'=0}^{\frac{P}{2}} \omega_{\frac{P}{2}}^{lj'} x(2j'+1), \quad \forall l \in [0, \frac{P}{2} - 1]$$

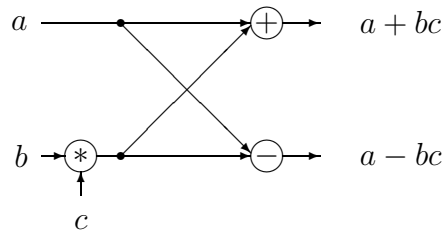


Figure 4: Decimation-in-time butterfly.

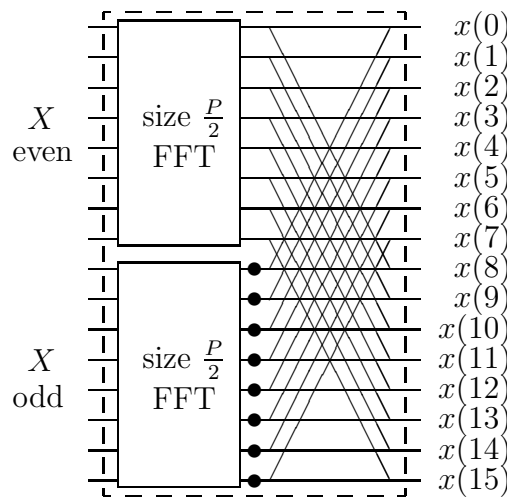


Figure 5: Decimation-in-time decomposition of the discrete Fourier Transform.

since  $\omega_P^{l+\frac{P}{2}} = (-1)\omega_P^l$ . The formulas

$$X(l) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{lj'} x(2j') + \omega_P^l \sum_{j'=0}^{\frac{P}{2}} \omega_{\frac{P}{2}}^{lj'} x(2j' + 1), \quad \forall l \in [0, \frac{P}{2} - 1],$$

$$X(l + \frac{P}{2}) = \sum_{j'=0}^{\frac{P}{2}-1} \omega_{\frac{P}{2}}^{lj'} x(2j') - \omega_P^l \sum_{j'=0}^{\frac{P}{2}} \omega_{\frac{P}{2}}^{lj'} x(2j' + 1), \quad \forall l \in [0, \frac{P}{2} - 1]$$

are the decimation-in-time splitting formulas. The DIT butterfly is shown in Figure 4. It differs from the DIF butterfly in that the complex multiplication of data with a twiddle factor appears on one of the inputs, instead on one of the outputs in the DIF butterfly. The effect on the complete computation of the splitting formula is shown in Figure 5.

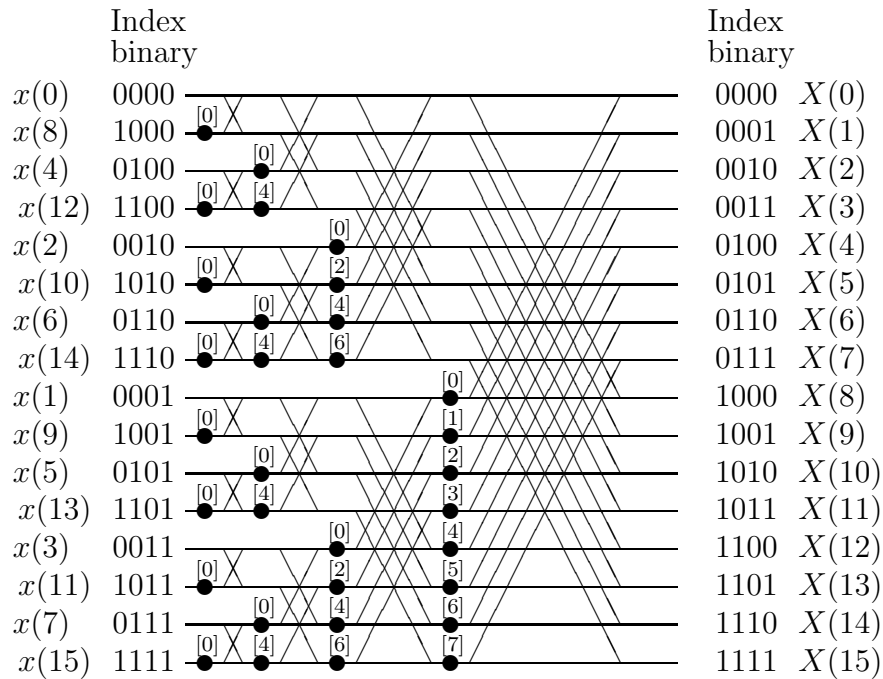


Figure 6: A 16-point decimation-in-time FFT with input in bit-reversed order.

Note that the derivation of the DIT FFT proceeds from the output towards the input, and that as the derivation proceeds the input eventually is required in bit-reversed order for the complete DIT FFT. The result is shown in Figure 6.

The recursive application of the splitting formulas yields a *bit-reversed output* order for a *normal order input* for the *DIF FFT*, while for the *DIT FFT* a *normal order output* requires a *bit-reversed order input*. The fundamental property of the FFT, whether DIF or DIT, is that it bit-reverses the order of the data from input to output, when an in-place algorithm is used. In fact, both DIF and DIT FFT can accept data in normal order with the output being in bit-reversed order for an in-place algorithm, or either algorithm can use bit-reversed input order and produce a result in normal order.

The normal order, or the bit-reversed order, describes the mapping between array indices and memory locations. Both the DIF and DIT FFT proceeds by computing butterflies on successively lower order bits in the array index in proceeding from input to output.

**Theorem 2** *The butterfly computations in the Cooley–Tukey FFT combines data that differ in successively lower order bits in their binary encoding in proceeding from input to output for both DIT and DIF FFT.*

The observation in Theorem 2 is very important since it clearly decouples the operations in the index space from the memory locations. Thus, a DIF FFT for a bit-reversed mapping of input

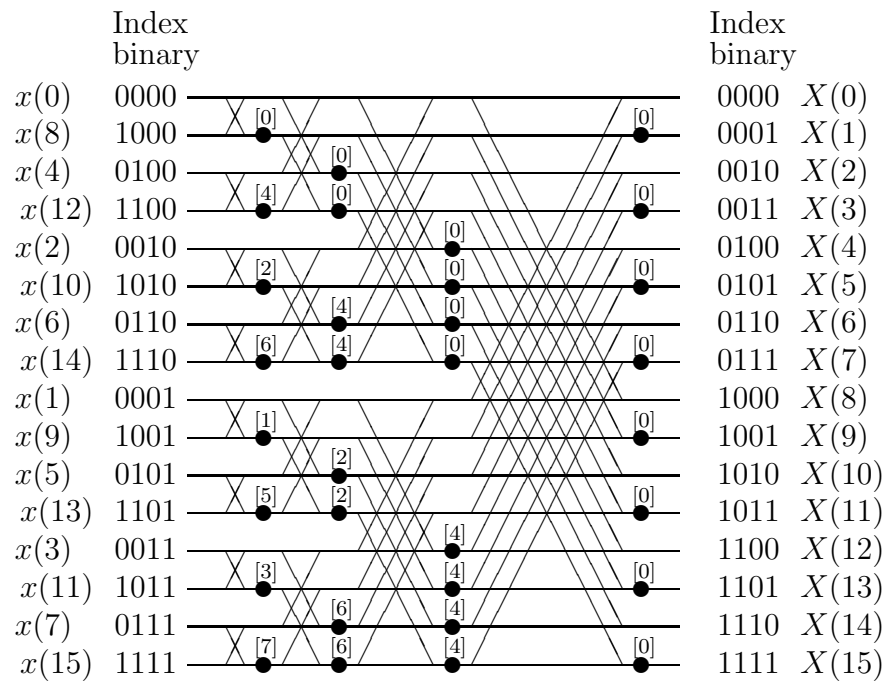


Figure 7: A 16–point decimation–in–frequency FFT with input in bit–reversed order.

data to memory addresses is obtained by simply bit–reversing the addresses used compared to the DIF on normal order input data. Figure 7 shows a DIF FFT with bit–reversed order input. The Figure is derived from Figure 3 by permuting the horizontal lines according to a bit–reversal of the addresses. Similarly, a DIT FFT for input data in normal order is also obtained by using the bit–reversed value of the address used for bit–reversed input order. Figure 8 shows a DIT FFT with normal order input. Figure 8 is obtained by permuting the rows in Figure 6 according to a bit–reversal of the addresses.

**Theorem 3** *The discrete Fourier Transform can be computed in  $5P \log_2 P - 9P + 12$  real arithmetic operations with either a DIT or a DIF radix–2 FFT on data with input order being either normal or bit–reversed.*

The most important difference between the DIF and DIT FFTs with respect to distributed memory systems is that the DIF and DIT FFTs use the twiddle factors in opposite order. From Figures 3 and 8 we notice that if a DIF FFT follows a DIF FFT without reordering between the two transforms, then the input to the DIF FFT is in bit–reversed order and the twiddle factors appear in the same order from top to bottom in the DIT and DIF transforms, but in reverse order from input to output. A DIF FFT on 16 points in bit–reversed input order is shown in Figure 7.

If the stages of the FFT are numbered from 0 to  $\log_2 N - 1$ , then

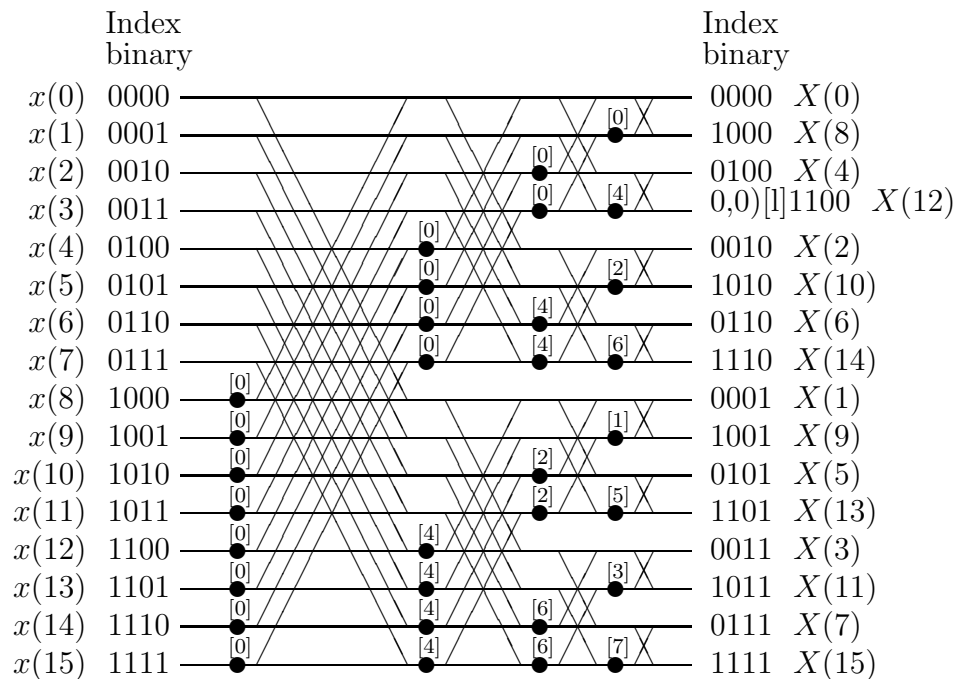


Figure 8: A 16–point decimation–in–time FFT with normal order input.

- in stage  $q$  of the DIF FFT, twiddle factors  $\omega_{\frac{P}{2^q}}^k, k = \{0, \dots, \frac{P}{2^{q+1}} - 1\}$  are used, each twiddle factor being used  $2^q$  times.
- in stage  $q$  of the DIT FFT, twiddle factors  $\omega_{2^{q+1}}^k, k = \{0, \dots, 2^q - 1\}$  are used, each twiddle factor being used  $\frac{P}{2^{q+1}}$  times.

The twiddle factors used in the last stage of the DIF FFT and the first stage of the DIT FFT are all equal to  $\omega_2^0 = 1$ .

This difference causes a difference in the demand for twiddle factor storage by a factor of  $\log_2 N$  for an  $N$  processor system, as discussed later.

## 1.2 High radix FFT

The splitting formulas above yield a radix–2 FFT. Each butterfly operates on two data points. A radix– $R$  FFT algorithm is a simple generalization of the radix–2 FFT algorithm such that each butterfly computation involves  $R$  data points instead of two points. Using a higher radix offers a small reduction in the number of arithmetic operations, but more importantly, a reduction in the number of data references between successive levels in a memory hierarchy with the proper scheduling of operations. The radix– $R$  FFT algorithms are derived by decomposing the computation of the Discrete Fourier Transform of size  $P$  into  $R$  subproblems of size  $\frac{P}{R}$  each,

instead of two subproblems of size  $\frac{P}{2}$ . The integer  $R$  is the radix. For Cooley–Tukey type FFT it is usually a small power of 2, such as 4 or 8.

### 1.2.1 Radix-4 FFT

For the derivation of a radix-4 DIF FFT  $j = j' + k\frac{P}{4}$ ,  $j' \in [0, \frac{P}{4} - 1]$ ,  $k \in [0, 3]$ .

$$\begin{aligned} X(l) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{lj'} x(j') + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+\frac{P}{4})} x(j' + \frac{P}{4}) + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+2\frac{P}{4})} x(j' + 2\frac{P}{4}) \\ &\quad + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(j'+3\frac{P}{4})} x(j' + 3\frac{P}{4}) \quad \forall l \in [0, P-1] \end{aligned}$$

Rewriting this expression with  $l = r + 4l'$ ,  $r \in [0, 3]$ ,  $l' \in [0, \frac{P}{4} - 1]$  yields

$$\begin{aligned} X(4l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} (x(j') + x(j' + \frac{P}{4}) + x(j' + 2\frac{P}{4}) + x(j' + 3\frac{P}{4})) \\ X(4l' + 1) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{j'} (x(j') - ix(j' + \frac{P}{4}) - x(j' + 2\frac{P}{4}) + ix(j' + 3\frac{P}{4})) \\ X(4l' + 2) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{2j'} (x(j') - x(j' + \frac{P}{4}) + x(j' + 2\frac{P}{4}) - x(j' + 3\frac{P}{4})) \\ X(4l' + 3) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{3j'} (x(j') + ix(j' + \frac{P}{4}) - x(j' + 2\frac{P}{4}) - ix(j' + 3\frac{P}{4})). \end{aligned}$$

These expressions can be reorganized as follows [14, 15].

$$\begin{aligned} X(4l') &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} ([x(j') + x(j' + 2\frac{P}{4})] + [x(j' + \frac{P}{4}) + x(j' + 3\frac{P}{4})]) \\ X(4l' + 1) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{j'} ([x(j') - x(j' + 2\frac{P}{4})] - i[x(j' + \frac{P}{4}) - x(j' + 3\frac{P}{4})]) \\ X(4l' + 2) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{2j'} ([x(j') + x(j' + 2\frac{P}{4})] - [x(j' + \frac{P}{4}) + x(j' + 3\frac{P}{4})]) \\ X(4l' + 3) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} \omega_P^{3j'} ([x(j') - x(j' + 2\frac{P}{4})] + i[x(j' + \frac{P}{4}) - x(j' + 3\frac{P}{4})]). \end{aligned}$$

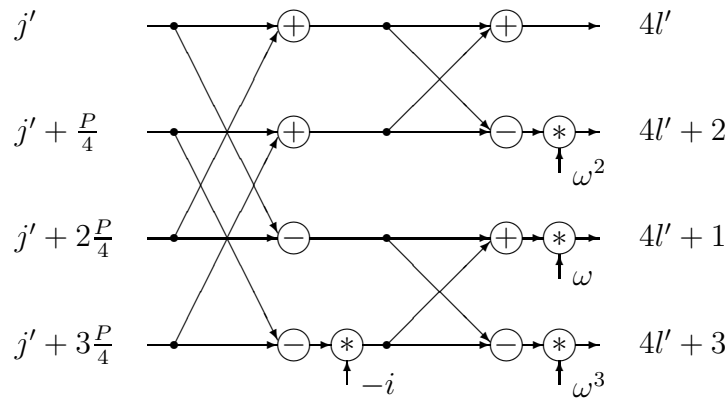


Figure 9: Factoring of a radix-4 DIF butterfly.

The latter set of equations show that a radix-4 stage can be organized as two radix-2 stages, as shown in Figure 9. The derived algorithm is a radix-4 DIF FFT. We note that complex multiplication for the radix-4 butterfly is required only at the output stage. Thus, a total of three complex multiplications are required, compared to four complex multiplications for two stages of a radix-2 FFT. Furthermore, if the intermediate data is held in temporary storage, then only half as many references to memory is required for a radix-4 FFT compared to a radix-2 FFT.

Reviewing Figure 9 carefully it is very important to observe one detail: the output is bit-reversed. Not only are all even data ordered before all odd data, but within the group of even data all even data with respect to the second least significant bit are ordered before the odd data with respect to that bit. The same ordering is true for the odd data. Thus, the first DIF radix-4 stage bit-reverses the two least significant bits and makes them the two most significant bits in the output ordering.

A DIT FFT is derived by expressing the data index  $j$  as  $j = 4j' + k$ ,  $j' \in [0, \frac{P}{4} - 1]$ ,  $k \in [0, 3]$ . Then,

$$\begin{aligned}
 X(l) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j')} x(4j') + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+1)} x(4j'+1) + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+2)} x(4j'+2) \\
 &\quad + \sum_{j'=0}^{\frac{P}{4}-1} \omega_P^{l(4j'+3)} x(4j'+3) \quad \forall l \in [0, P-1].
 \end{aligned}$$

Rewriting this expression with  $l = r\frac{P}{4} + l'$ ,  $r \in [0, 3]$ ,  $l' \in [0, \frac{P}{4} - 1]$  yields

$$X(l') = \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{l'r} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j'+1)$$

$$\begin{aligned}
 & + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \\
 X\left(\frac{P}{4} + l'\right) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - i\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) \\
 & - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) + i\omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \\
 X\left(2\frac{P}{4} + l'\right) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) \\
 & + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \\
 X\left(3\frac{P}{4} + l'\right) &= \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + i\omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) \\
 & - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) - i\omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3)
 \end{aligned}$$

As in the case of DIF FFT this expression can be rewritten such that each radix-4 stage is implemented as two radix-2 stages.

$$\begin{aligned}
 X(l') &= \left[ \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) \right] \\
 &+ \left[ \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \right] \\
 X\left(\frac{P}{4} + l'\right) &= \left[ \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) \right] \\
 &- i \left[ \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \right] \\
 X\left(2\frac{P}{4} + l'\right) &= \left[ \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') + \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) \right] \\
 &- \left[ \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) + \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \right]
 \end{aligned}$$

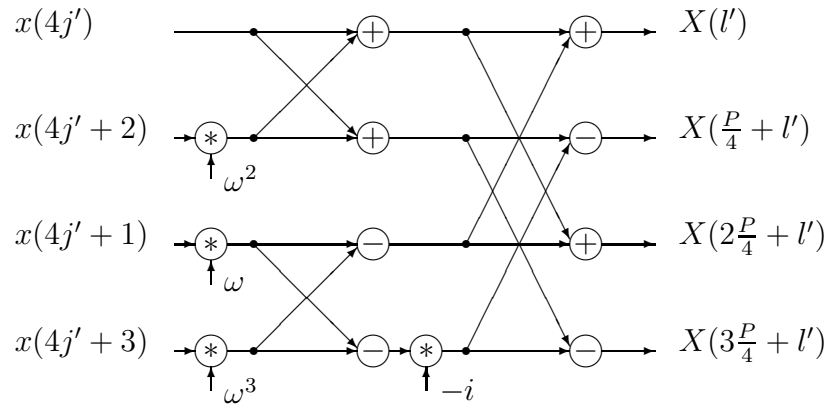


Figure 10: Factoring of a radix-4 DIT butterfly.

$$\begin{aligned}
 X\left(3\frac{P}{4} + l'\right) = & \left[ \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j') - \omega_P^{2l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 2) \right] \\
 & + i \left[ \omega_P^{l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 1) - \omega_P^{3l'} \sum_{j'=0}^{\frac{P}{4}-1} \omega_{\frac{P}{4}}^{l'j'} x(4j' + 3) \right].
 \end{aligned}$$

From Figure 10 it is clear that the radix-4 DIT FFT derivation results in an input order that is bit-reversed with respect to the output order, just as for the radix-2 DIT FFT. We have shown that the following result holds for radix-2 and radix-4 DIT and DIF FFT.

**Theorem 4** *The DIF and DIT in-place FFT produce an output ordering that is bit-reversed with respect to the input ordering, regardless of the type and radix of the FFT.*

**Corollary 1** *Reordering of the output of an FFT to be identical to the input order is independent of what radix was used in the transform, and independent of whether a DIF or DIT FFT was used.*

This observation is of great practical significance, since it allows for a complete decoupling of the reordering from the transform. The reordering can be made prior to the transform, integrated with the transform, or be performed after the transform.

From Figures 9 and 10 it is clear that complex multiplications are only required for every other radix-2 butterfly stage in both DIT and DIF radix-4 FFT. The multiplication by  $i$  requires no arithmetic operations, only an interchange of real and imaginary parts with the appropriate sign change.

The number of real arithmetic operations for a radix-4 FFT is  $\frac{17}{4}N \log_2 N - \frac{43}{6}N + \frac{32}{3}$  [15].

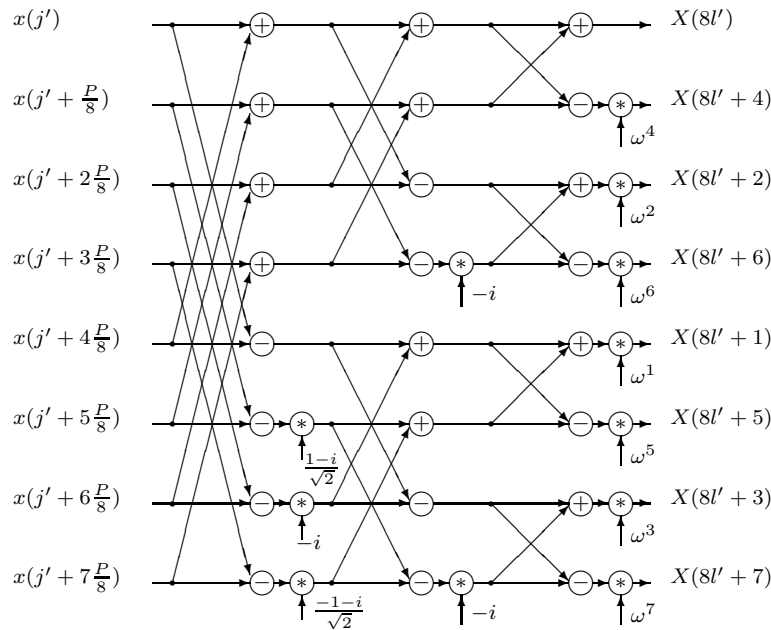


Figure 11: A decimation-in-frequency radix-8 kernel.

### 1.2.2 Radix-8 FFT

A radix-8 DIF FFT is derived by the factoring of  $j$  as  $j = j' + k\frac{P}{8}$ ,  $j' \in [0, \frac{P}{8} - 1]$ ,  $k \in [0, 7]$ , and by factoring  $l$  as  $l = r + 8l'$ ,  $r \in [0, 7]$ ,  $l' \in [0, \frac{P}{8} - 1]$ . A radix-8 DIT FFT can be derived similarly. In a radix-8 FFT, twiddle factor multiplications are only required for every three radix-2 butterfly stages, as seen in Figures 11 and 12.

### 1.3 Comparison of different radix Cooley-Tukey FFT

Different radix FFTs are merely reorganizations of the arithmetic operations. The required number of complex multiplications are reduced by factoring the matrix of twiddle factors  $W$  in different ways. High radix FFT reduces the number of stages where twiddle factors are needed, but as the radix increases, rotation factors internal to the high radix butterfly requires complex multiplications as well. A radix-2 butterfly uses internal twiddle factors corresponding to 180-degree rotations, radix-4 butterflies require 90-degree rotations, radix-8 butterflies 45-degree rotations, etc.

The fact that the FFT can be viewed as matrix factorization has been observed by many authors. An excellent treatment is given in [19]. The idea of the matrix factorization view is illustrated as follows for the DIF radix-2 FFT

$$W_P = \Pi \begin{bmatrix} W_{\frac{P}{2}} & 0 \\ 0 & W_{\frac{P}{2}} \end{bmatrix} \begin{bmatrix} I & I \\ D_P & D_P \end{bmatrix} x,$$

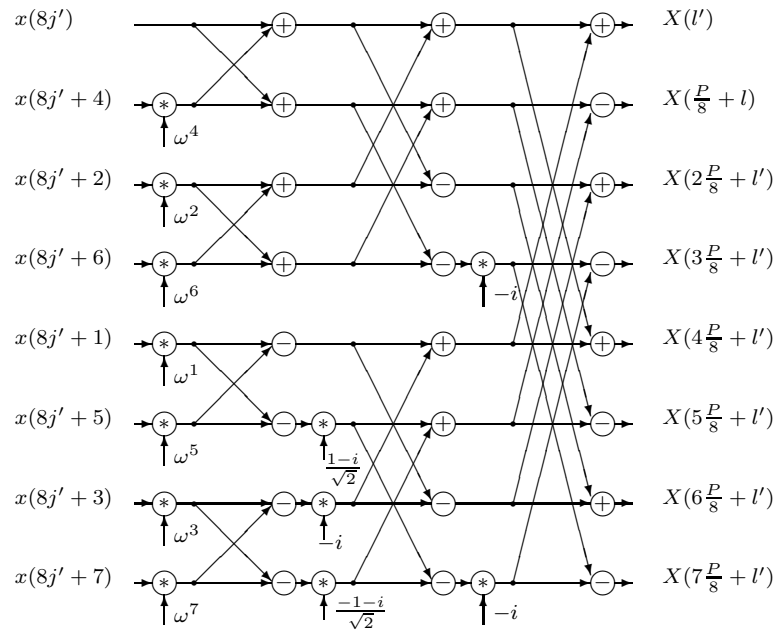


Figure 12: A decimation-in-time radix-8 kernel.

where  $\Pi$  is a permutation matrix,  $I$  the identity matrix,  $D_P$  a diagonal matrix of twiddle factors, and  $O$  a matrix with all zeroes.

The number of complex multiplications for a radix-2 FFT is  $(p - 1)\frac{P}{2}$ , while for a radix-4 FFT it is  $\frac{p}{2}\frac{3P}{4}$ . A radix-8 algorithm requires internal complex multiplications corresponding to 45-degree rotations. The number of real operations (considering higher order terms only) and memory accesses for radix-2, -4, and -8 butterflies are given in Table 3. The total number of operations are summarized in Table 4. The ratios of the number of real arithmetic operations normalized to the radix-8 FFT are  $\frac{60}{49} : \frac{51}{49} : 1$ . The total number of arithmetic operations for the radix-8 algorithm is approximately 20% less than that of the radix-2 algorithm. The exact number of multiplications and additions can be found, for instance, in [15].

In most architectures the effective use of the memory bandwidth is more critical with respect to performance than minimizing the number of arithmetic operations. With a local memory of size  $M$ , a radix- $M$  FFT offers a reduction in memory bandwidth requirement by a factor of  $\log_M$ , which is optimum [5]. The ratios for memory operations are  $\frac{60}{23} : \frac{33}{23} : 1$ . However, it should be noticed that if  $R = 2^r$  registers (complex) are used for twiddle factors, then  $r$  ranks can be computed with a single load of  $R - 1$  twiddle factors by successively computing all butterflies in a given rank requiring those coefficients. The adjacent set of  $r$  radix-2 butterfly ranks require  $R$  loads of  $R - 1$  twiddle factors, etc. The total number of storage references for twiddle factors is  $P - 1$ . If the ordering of the computations is such that  $P - 1$  loads of twiddle factors suffice, then the storage references for data dominate. The ratios of the number of memory references become  $3 : \frac{3}{2} : 1$  for radix-2, -4, and -8 FFT. The number of registers needed for twiddle factors are 2, 6, and 14, respectively.

FFT	Arithmetic Operations			Storage References		
	Add/Sub	Mult	Total	Data	Twiddles	Total
Radix-2	6	4	10	8	2	10
Radix-4	22	12	34	16	6	22
Radix-8	66	32	98	32	14	46

Table 3: Arithmetic and memory operations for radix-2, -4, and -8 butterflies.

FFT	Arithmetic Operations			Storage References		
	Add	Mult	Total	Data	Twiddles	Total
Radix-2	$3Pp$	$2Pp$	$5Pp$	$4Pp$	$Pp$	$5Pp$
Radix-4	$\frac{22}{8}Pp$	$\frac{12}{8}Pp$	$\frac{17}{4}Pp$	$\frac{16}{8}$	$\frac{6}{8}Pp$	$\frac{11}{4}$
Radix-8	$\frac{66}{24}Pp$	$\frac{32}{24}Pp$	$\frac{49}{12}Pp$	$\frac{32}{24}Pp$	$\frac{14}{24}Pp$	$\frac{23}{12}Pp$

Table 4: Arithmetic and memory operations for radix-2, -4, and -8 FFTs.

As an example of the results of using a combination of radix-2, radix-4 and radix-8 kernels Figure 13 shows the Connection Machine CM-2 performance for FFT local to a node. The radix for the node is limited by the number of registers per node, which is 32.

## 1.4 Twiddle factors

### 1.4.1 Radix-2 FFT

The total number of twiddle factors needed for a radix-2 FFT of size  $P = 2^p$  is  $\frac{P}{2} - 1$ .

For the DIF FFT all twiddle factors  $\omega_P^j = e^{-\frac{2\pi i}{P}j}$ ,  $j \in [0, \frac{P}{2} - 1]$  are used in the first rank. As the computations proceed from the input to the output, the number of distinct twiddle factors needed decreases. For the radix-2 DIF FFT, the exponent of the twiddle factor for the butterfly on elements  $x(j)$  and  $x(j + \frac{P}{2})$  is  $(j_{p-1}) \times (j_{p-2}j_{p-3} \dots j_0)$ , where  $j = (j_{p-1}j_{p-2} \dots j_0)$ . For the second rank the exponent of the twiddle factor  $\omega_P$  is  $(j_{p-2}) \times (j_{p-3}j_{p-4} \dots j_0)$  for the pair of elements  $x(j)$  and  $x(j + \frac{P}{4})$ . In general, for a DIF radix-2 FFT [16], the twiddle factor required for the computation of a butterfly on the elements  $x(j)$  and  $x(j + \frac{P}{2^{q+1}})$  in stage  $q \in [0, p - 1]$  is  $\omega_P^{(j_{p-q-1}) \times (j_{p-q-2}j_{p-q-3} \dots j_0) 2^q}$ . The first butterfly stage is stage zero. The two data indices in a butterfly computation share all bits but the one that corresponds to the butterfly stage. The exponent of the twiddle factor is simply the shared lower part of the binary encoding of the data indices, i.e., bits  $p - q - 2$  through bit 0, shifted left  $q$  steps with an end-off shift.

This computation is particularly simple if the data is allocated in normal order upon input to

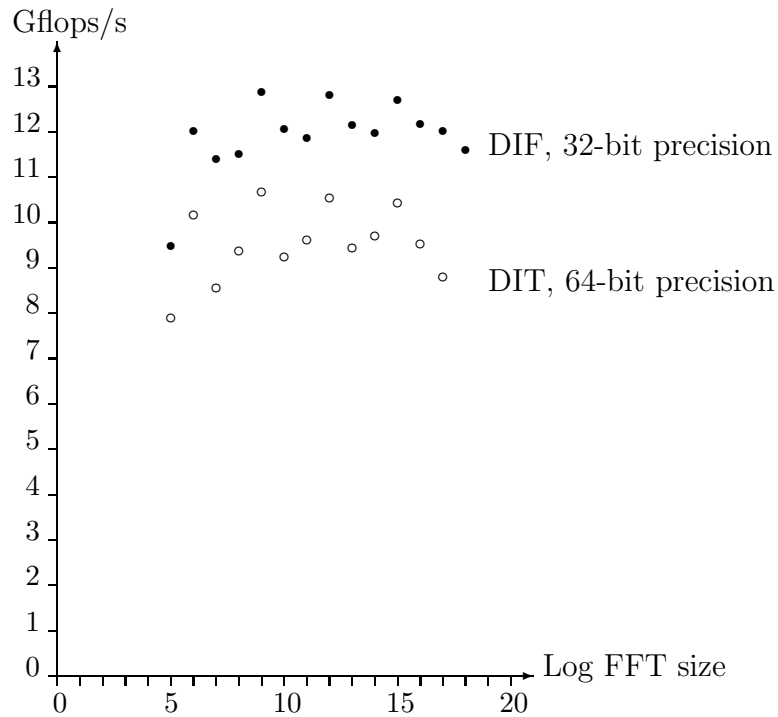


Figure 13: The performance of local, unordered, DIF CCFFT on a 2048 processor CM-200.

the DIF FFT. In this case, the index of an element corresponds to its location relative to the beginning of the array, and the relative array address can be used for the computation of the twiddle exponents in an *in-place* radix-2 DIF FFT.

For a DIT radix-2 FFT, the twiddle factors for stage  $q$  are  $\omega_p^{(j_{p-q-1})(j_{p-q}j_{p-q+1}\dots j_{p-1})2^{p-1-q}}$ ,  $q \in [0, p - 1]$ . For an *in-place* DIT FFT with input data in normal order, a fraction of the bit-reversed address properly shifted is used for the exponent (instead of a straight fraction of the address for the DIF FFT).

If data is allocated in a bit-reversed order, then array addresses must be bit-reversed for use in twiddle exponent computations in the DIF FFT, while normal addresses are used for the DIT FFT. The reason is that what really is required is the relevant fraction of the data index. Moreover, we see that DIF FFT in normal order and DIT FFT with data in bit-reversed order both use addresses in normal order for twiddle exponent computations. Similarly, DIT FFT in normal order and DIF FFT with input data in bit-reversed order use array addresses in the same way (bit-reversed) for twiddle exponent computation. This observation is particularly critical for distributed FFT using precomputed twiddle factors.

### 1.4.2 High Radix FFT

The total number of twiddle factors needed for a radix- $R$  FFT of size  $P$  is  $(R - 1)\frac{P}{R}$ .

For the computation of the twiddle factor exponent of high radix FFT we use an iterative

formulation of the FFT. We start with the radix-2 DIF FFT, then generalize the expressions to higher radix DIF FFT. The radix-2, in-place, DIF FFT can be formulated as

$$\begin{aligned}
 \tilde{x}_{-1}(a_{p-1}, \dots, a_0) &= x(a_{p-1}, \dots, a_0) \\
 \tilde{x}_0(a_{p-1}, \dots, a_0) &= (\tilde{x}_{-1}(0, a_{p-2}, \dots, a_0) + \omega_2^{a_{p-1}} \tilde{x}_0(1, a_{p-2}, \dots, a_0)) \omega_P^{\langle a_{p-2}, \dots, a_0 \rangle a_{p-1}} \\
 \tilde{x}_1(a_{p-1}, \dots, a_0) &= (\tilde{x}_0(a_{p-1}, 0, a_{p-3}, \dots, a_0) + \omega_2^{a_{p-2}} \tilde{x}_0(a_{p-1}, 1, a_{p-3}, \dots, a_0)) \omega_{\frac{P}{2}}^{\langle a_{p-3}, \dots, a_0 \rangle a_{p-2}} \\
 &\vdots \\
 \tilde{x}_q(a_{p-1}, \dots, a_0) &= (\tilde{x}_{q-1}(a_{p-1}, \dots, 0_{p-q-1}, \dots, a_0) + \omega_2^{a_{p-q-1}} \tilde{x}_{q-1}(a_{p-1}, \dots, 1_{p-q-1}, \dots, a_0)) \omega_{\frac{P}{2^q}}^{\langle a_{p-q-2}, \dots, a_0 \rangle a_{p-q-1}} \\
 &\vdots \\
 \tilde{x}_{p-1}(a_{p-1}, \dots, a_0) &= (\tilde{x}_{p-2}(a_{p-1}, \dots, a_1, 0) + \omega_2^{a_0} \tilde{x}_{p-2}(a_{p-1}, \dots, a_1, 1)) \omega_2^{0a_0} \\
 X(a_{p-1}, \dots, a_0) &= \tilde{x}_{p-1}(a_0, \dots, a_{p-1})
 \end{aligned}$$

The iterative formulation can be generalized to radix- $2^r$  FFT. With  $s \in [0, u-1]$  ( $u = r$ ), and  $(d_{u-1}d_{u-2} \dots d_0)$  the indices expressed in base  $R = 2^r$ , the radix- $2^r$ , in-place, DIF FFT can be written as

$$\begin{aligned}
 \tilde{x}_{-1}(d_{u-1}, \dots, d_0) &= x(d_{u-1}, \dots, d_0) \\
 \tilde{x}_s(d_{u-1}, \dots, d_0) &= \omega_{\frac{P}{R^s}}^{\langle d_{u-s-2}, \dots, d_0 \rangle \widehat{d_{u-s-1}}} \sum_{j=0}^{R-1} \tilde{x}_{s-1}(d_{u-1}, \dots, d_{u-s}, j, d_{u-s-2}, \dots, d_0) \omega_R^{\widehat{d_{u-s-1}}j} \\
 \tilde{x}_{u-1}(d_{u-1}, \dots, d_0) &= \omega_{\frac{P}{R^{u-1}}}^{\langle 0 \rangle \widehat{d_0}} \sum_{j=0}^{R-1} \tilde{x}_{u-2}(d_{u-1}, \dots, d_1, j) \omega_R^{\widehat{d_0}j} \\
 X(\widehat{d_{u-1}}, \dots, \widehat{d_0}) &= \tilde{x}_{u-1}(d_0, \dots, d_{u-1})
 \end{aligned}$$

where the bit-reversed value of a digit  $d_i$  is  $\widehat{d}_i$ .

For a radix- $2^r$ , in-place, DIF FFT with normal order input the required twiddle factor exponent for data with index  $(j_{p-1}j_{p-2} \dots j_0) = (d_{u-1}d_{u-2} \dots d_0)$  is  $\widehat{d_{u-1}} \times (d_{u-2}d_{u-3} \dots d_0)$  after the first radix- $2^r$  stage. For the second radix- $2^r$  stage the set of twiddle factor exponents are  $\widehat{d_{u-2}} \times (d_{u-3}d_{u-4} \dots d_0)2^r$ . The first factor in a twiddle factor index is the *bit-reversed* value of a radix- $2^r$  digit defined by  $r$  consecutive bits in the binary encoding of the data index. For the first radix- $2^r$  stage the digit is defined by the  $r$  most significant index bits, for the second radix- $2^r$  stage the next  $r$  bits in the index encoding are used, etc. The second factor in the twiddle factor index is defined by all data index bits of lower order than the radix- $2^r$  digit used for the first factor. The twiddle factor index is obtained by multiplying the product of the two factors obtained from the data index by  $2^r$  as many times as corresponds to the radix- $2^r$  stage, with the first stage being stage zero.

In general, for a radix- $2^r$  *in-place* DIF FFT algorithm, the addresses of the data can be used instead of the initial index to determine the twiddle factor exponents, just as in the radix-2 case.

Thus, with addresses denoted  $(a_{p-1}a_{p-2} \dots a_0)$ , the twiddle factor index required for the data item in location  $(a_{p-1}a_{p-2} \dots a_0)$  after the  $s$ th radix- $2^r$  stage is  $d_{u-s-1} \times (d_{u-s-2}d_{u-s-3} \dots d_0)2^{sr}$ .

High radix DIT FFT requires twiddle factor application (complex multiplications) to the input to stages  $q \bmod r$ , where  $q$  is the radix-2 stage. As for the DIF FFT, we express the twiddle factor exponents for an in-place algorithm in terms of memory addresses  $(a_{p-1}a_{p-2} \dots a_0)$  and stage number. The iterative formulation of a DIT radix-2 FFT is

$$\begin{aligned}
 \tilde{x}_{-1}(a_{p-1}, \dots, a_0) &= x(a_{p-1}, \dots, a_0) \\
 \tilde{x}_0(a_{p-1}, \dots, a_0) &= \tilde{x}_{-1}(0, a_{p-2}, \dots, a_0) + \omega_2^{a_{p-1}} \omega_2^0 \tilde{x}_{-1}(1, a_{p-2}, \dots, a_0) \\
 \tilde{x}_1(a_{p-1}, \dots, a_0) &= \tilde{x}_0(a_{p-1}, 0, a_{p-3}, \dots, a_0) + \omega_2^{a_{p-2}} \omega_4^{\langle a_{p-1} \rangle} \tilde{x}_0(a_{p-1}, 1, a_{p-3}, \dots, a_0) \\
 &\vdots \\
 \tilde{x}_s(a_{p-1}, \dots, a_0) &= \tilde{x}_{s-1}(a_{p-1}, \dots, 0_{p-s-1}, \dots, a_0) + \omega_2^{a_{p-s-1}} \omega_{2^{s+1}}^{\langle a_{p-s}, \dots, a_{p-1} \rangle} \tilde{x}_{s-1}(a_{p-1}, \dots, 1_{p-s-1}, \dots, a_0) \\
 &\vdots \\
 \tilde{x}_{p-1}(a_{p-1}, \dots, a_0) &= \tilde{x}_{p-2}(a_{p-1}, \dots, a_1, 0) + \omega_2^{a_0} \omega_P^{\langle a_1, \dots, a_{p-1} \rangle} \tilde{x}_{p-2}(a_{p-1}, \dots, a_1, 1) \\
 X(a_{p-1}, \dots, a_0) &= \tilde{x}_{p-1}(a_0, \dots, a_{p-1})
 \end{aligned}$$

A radix- $2^r$ , in-place, DIT FFT can be written as

$$\begin{aligned}
 \tilde{x}_{-1}(d_{u-1}, \dots, d_0) &= x(d_{u-1}, \dots, d_0) \\
 \tilde{x}_s(d_{u-1}, \dots, d_0) &= \sum_{j=0}^{R-1} \omega_R^{d_{u-s-1}j} \omega_{R^{s+1}}^{\langle d_{u-s}, \dots, d_{u-1} \rangle j} \tilde{x}_{s-1}(d_{u-1}, \dots, d_{u-s}, j, d_{u-s-2}, \dots, d_0) \\
 \tilde{x}_{u-1}(d_{u-1}, \dots, d_0) &= \sum_{j=0}^{R-1} \omega_R^{\widehat{d}_0 j} \omega_N^{\langle \widehat{d}_1, \dots, \widehat{d}_{u-1} \rangle j} \tilde{x}_{u-2}(d_{u-1}, \dots, d_1, j) \\
 X(\widehat{d}_{u-1}, \dots, \widehat{d}_0) &= \tilde{x}_{u-1}(d_0, \dots, d_{u-1})
 \end{aligned}$$

The indices of the twiddle factors for normal order input are all one for the first stage,  $j \times \widehat{d}_{u-1} 2^{p-2r}$  for the second radix- $2^r$  stage, and  $j \times (\widehat{d}_{u-s} \dots \widehat{d}_{u-1}) 2^{p-(s+1)r}$  for an arbitrary stage  $s$ . Note, that the address is bit-reversed and shifted for the proper exponent when the data is allocated in normal order, just as in the radix-2 case.

### 1.4.3 Bit-reversed input

The FFT computations always proceed from the most significant bit to the least significant bit in the index space, whether a DIF or DIT is used, and regardless of the radix being used. With the input in bit-reversed order the traversal of the bits in the address field is from the lowest order to the highest order bit, since the most significant index bit is mapped to the least significant address bit with input data in bit-reversed order.

With respect to twiddle exponent computation, bit-reversed address must be used wherever a normal index encoding is desired, and normal addresses used wherever bit-reversed indices are required.

## 1.5 Summary of Cooley–Tukey FFT

The most important properties of the Cooley–Tukey FFT in addition to its low arithmetic count compared to matrix–vector multiplication are

- the butterfly computations proceed from the most significant to the least significant bit in the data index encoding for *both* DIT and DIF FFT,
- the DIF and DIT FFT in-place algorithms both produce results that are bit-reversed with respect to the input ordering,
- the bit-reversal property in the DIF and DIT FFT is independent of the radix
- the DIF and DIT FFT use the same twiddle factors, but in opposite order from input to output,
- a radix- $R$  FFT reduces the demand for memory bandwidth by a factor of  $R$  if all temporary variables are held in registers (local memory)

## 2 Parallel FFT implementations

The FFT is a computationally very efficient algorithm. In a parallel computing system it tends to be quite sensitive to the performance of the communication system. The computation of an FFT on  $M$  elements requires about  $5M \log_2 M$  operations. Thus, if the memory holds  $M$  elements,  $2M$  cycles are required to load the data assuming one word can be loaded each cycle (since the data is complex). Similarly,  $2M$  cycles are required to store the data. In addition, cycles are either required to compute twiddle factors, or to load twiddle factors. For the latter case,  $M/2$  complex twiddle factors must be loaded in  $M$  cycles. (In fact, the number of twiddle factors loaded can be reduced without requiring additional arithmetic by using 90-degree rotations, or possibly also 45-degree rotations, as discussed later.) Thus, the ratio of the number of arithmetic operations to the number of load/store cycles is  $\log_2 M$ , assuming that  $M$  cycles are used to load twiddle factors. This strategy for computing FFTs on uni-processors was suggested by Gentleman and Sande [3] and proved to be optimal by Hong and Kung [5]. The idea was later adopted to parallel computing by Johnsson et. al. [10, 11] and Swartztrauber [17].

Computing the FFT based on radix- $M$  FFT as outlined above requires the equivalent of matrix transposition between successive radix- $M$  stages for the computation of an FFT on  $P \gg M$  points, as described below. The sequence of matrix transpositions introduces a permutation of the data in addition to the bit-reversal of the FFT. We will show that the permutation

caused by the sequence of matrix transpositions is an *unshuffle* on a part of the index space. An *ordered* transform, i.e., a transform in which the index order for input and output are the same, requires a shuffle and a bit-reversal that may be combined into one permutation.

A straightforward parallelization of the Cooley–Tukey FFT in which variables are computed just as in the sequential case only introduces the need for a bit-reversal for an ordered transform. The arithmetic load–balance is almost as good as an algorithm based on permutations, but not perfect. An advantage of the straightforward implementation is that indirect addressing is not required, while the algorithm employing permutations does require indirect addressing, albeit of a simple form.

Since the twiddle factors only depend upon the size of the data set, but do not depend on the data itself, they can be precomputed and used for all FFT of a given size. The twiddle factor computation is time consuming compared to addition and multiplication since it involves the computation of *sine* and *cosine* functions. Multiple transforms of the same size occurs frequently in performing Fourier Transforms on higher dimensional arrays. For instance, in a three–dimensional array, two of the axes defines the number of copies of transforms to be performed along the third axis.

The twiddle factor storage is of particular concern for distributed FFT with precomputed twiddle factors. We will show that with proper choice of algorithm selection (DIT or DIF) the storage requirement for no communication of twiddle factors need not be more than a factor of two greater than the storage requirement on a single processor. On the other hand, with a poor choice, either the twiddle factor storage may be a factor of  $O(\log_2 N)$  higher than in the sequential case, i.e., a factor of  $O(\log_2 N)$  more space is required for twiddle factors than for data, or communication of twiddle factors is required.

Next we review cyclic and consecutive index allocation. Then, we discuss the straightforward parallelization of the Cooley–Tukey FFT, followed by a description of permutation based FFT in the form of bi–section and multi–section FFT.

## 2.1 Data Allocation

In the *consecutive* data allocation [6] a number of successive elements are allocated to the same memory module. Assuming that the number of processing nodes  $N = 2^n$ ,  $n$  address bits are assigned to the encoding of node addresses. The mapping of the array indices to machine addresses for  $P = 2^p$  data elements can be viewed as follows, where  $x_i$  denotes a bit in the encoding of the data indices:

*Consecutive* assignment:

$$\underbrace{(x_{p-1} \dots x_{p-n})}_{paddr} \underbrace{x_{p-n-1} x_{p-n-2} \dots x_0}_{maddr}.$$

The field denoted *paddr* encodes *node* addresses as opposed to *memory* addresses, *maddr*. In *cyclic* assignment the lowest order bits in the encoding of array indices are mapped to the node address field.

Consecutive data allocation								Cyclic data allocation							
$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
0	4	8	12	16	20	24	28	0	1	2	3	4	5	6	7
1	5	9	13	17	21	25	29	8	9	10	11	12	13	14	15
2	6	10	14	18	22	26	30	16	17	18	19	20	21	22	23
3	7	11	15	19	23	27	31	24	25	26	27	28	29	30	31

Figure 14: Consecutive and cyclic data allocation of 32 elements to 8 nodes.

Input order	Consecutive	Cyclic
Normal	First $n$ stages	Last $n$ stages
Bit-reversed	Last $n$ stages	First $n$ stages

Table 5: The relationship between input order, data allocation, and stages requiring communication for distributed FFT.

*Cyclic* assignment:

$$\underbrace{(x_{p-1}x_{p-2} \dots x_n)}_{maddr} \underbrace{x_{n-1}x_{n-2} \dots x_0}_{paddr}.$$

All data elements with the same  $n$  low order bits reside in the same node. In the consecutive assignment the indices of all elements in a node have the same  $n$  high order bits. The consecutive and cyclic allocations of a 32 element one-dimensional array among 8 nodes are illustrated in Figure 14. We consider the impact of these forms of data allocation on the data motion requirements for the FFT.

For multidimensional arrays, we assume that each axis is encoded separately, as for instance is the case in the Connection Machine programming systems [18] and High Performance Fortran [2]. Note that HPF supports both cyclic and consecutive index allocation.

We note that for normal order input, the first  $n$  radix-2 stages require internode communication in a consecutive data allocation. The last  $p - n$  radix-2 stages are local to a node. For a cyclic data allocation and normal input order, the first  $p - n$  radix-2 stages are local, while the last  $n$  stages require communication. If the input order is bit-reversed, then for the consecutive order the first  $p - n$  stages are local to a node, while the last  $n$  radix-2 stages require communication. Analogously, bit-reversed input order and cyclic allocation results in communication for the first  $n$  stages, while the last  $p - n$  stages are local to each node. These observations are summarized in Table 5.

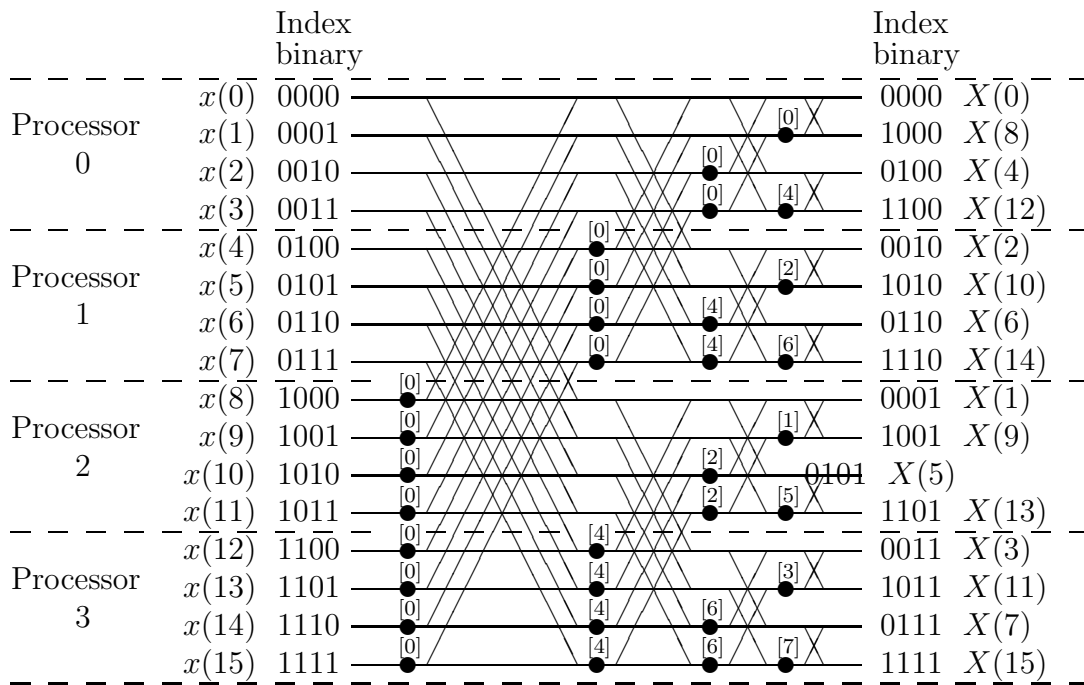


Figure 15: Decimation-in-time FFT on four processors. Normal order, consecutive data allocation.

## 2.2 Radix-2 FFT without explicit permutations

A straightforward parallelization of the FFT requires an exchange of data between nodes holding data for a butterfly. For instance, consider the use of a DIT FFT for consecutive data allocation, as illustrated in Figure 15. The first  $n$  stages, i.e., the first two stages in the Figure, requires communication for the computation of butterflies, while the last  $p - n$  stages involve only local data. For the first butterfly stage, processors differing in their highest order bit exchange data. Then, one processor in a pair computes the “top” of the butterfly, while the other computes the “bottom”. Thus all processors participates in the computation of the butterflies. An arithmetic load imbalance results from the fact that the “top” does not require a complex multiplication, while the “bottom” does require complex multiplication. Processors computing the “top” perform two arithmetic operations per butterfly, while the processors computing the “bottom” performs eight arithmetic operations. The load-balance can be improved by, for instance, having the two processors involved in a butterfly computation split the computations required for the complex multiplication, at the expense of extra communications.

The number of complex variables a processor must send and receive for each nonlocal stage of the FFT is  $\frac{P}{N}$  for the algorithm outlined above. The total number of complex variables exchanged for the complete FFT is  $n\frac{P}{N}$ . Balancing the arithmetic load increases the communication with a small factor.

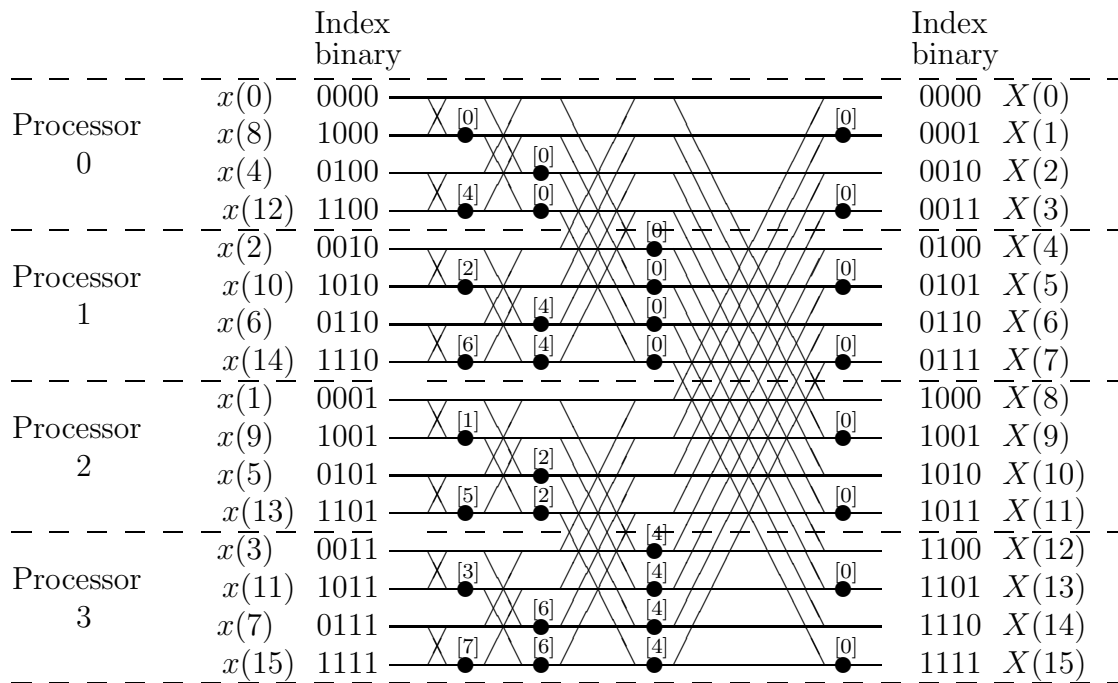


Figure 16: Decimation-in-frequency FFT with input in bit-reversed order.

The number of exchanged variables is the same for a radix-2 DIF FFT as for a radix-2 DIT FFT, as can be inferred from Figure 16. In Figure 16 the input data is in bit-reversed order and the data allocation is consecutive. By comparing Figures 15 and 16 it is clear that a forward DIT FFT followed by an inverse DIF FFT not only require no data reordering between the forward and inverse FFT, but also the twiddle factors used in the forward can also be used for the inverse FFT in each of the nodes. We will discuss this property further when we examine the twiddle factors in more detail.

The algorithms for the radix-2 DIT and DIF FFT outlined above are implemented on the Connection Machine systems CM-2/200 [12]. Improving the arithmetic load-balance at the expense of additional communication would yield lower performance on this computer given the applicable ratio of communication and computation.

### 2.3 Radix- $R$ FFT without explicit permutations

For a radix- $R$  FFT without explicit permutations and without a factoring of the radix- $R$  butterfly into radix-2 stages, *all-to-all broadcast* among disjoint sets of  $R$  nodes is required for each butterfly computation. After the all-to-all broadcast, each processor can compute the output of the radix- $R$  butterfly it shall store. The total communication for each node required by this approach is

Proc. id	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
initial	0	2	4	6	8	10	12	14
alloc.	1	3	5	7	9	11	13	15
after 1st exch.	0	2	4	6	1	3	5	7
	8	10	12	14	9	11	13	15
after 2nd exch.	0	2	8	10	1	3	9	11
	4	6	12	14	5	7	13	15
after 3rd exch.	0	4	8	12	1	5	9	13
	2	6	10	14	3	7	11	15
after 4th exch.	0	4	8	12	2	6	10	14
	1	5	9	13	3	7	11	15

Figure 17: The data distribution for a radix-2 FFT based on bi-section with normal order consecutive data allocation.

$$\left( (R-1) \lfloor \frac{n}{r} \rfloor + 2^{n \bmod r} - 1 \right) \frac{P}{N},$$

where  $R = 2^r$ . This expression is minimized for  $r = 1$ . Hence, carrying out a radix- $R$  FFT in this manner is not advantageous with respect to communication. Radix- $R$  FFT shall be factored into radix-2 FFT stages with respect to communication for distributed FFT.

## 2.4 Distributed FFT using explicit permutations

An alternate approach to the parallel FFT algorithms described above is to move the data such that a processor after the data motion has all data required for a butterfly computation. For instance, if each node has two complex data elements, then the processors can exchange data such that each processor after the exchange computes a complete butterfly using only local data. The arithmetic is perfectly load-balanced. The idea is illustrated in Figure 17 for normal order, consecutive data allocation.

For the normal order consecutive allocation an exchange of data between processors differing in their addresses on the most significant bit results in data differing on the most significant bit being local to a processor. Thus, the first stage of the FFT can be computed. The next stage requires a new exchange operation, etc. With more than one pair of elements per node, the same exchange discipline is applied to all pairs of local data, as discussed further below. We refer to a data allocation allowing for a single FFT stage per reallocation as a *bi-section* algorithm, while a reallocation allowing for multiple FFT stages to be performed is referred to as *multi-section*.

### 2.4.1 Bi-section

Figure 18 illustrates the use of bi-section when there are more than two data elements in node. The exchange operation in the first step is performed on the most significant local memory address bit and the most significant processor address bit. Subsequent stages still use the most significant memory bit for the exchange, while successively lower order processor address bits are used for the exchange. After the exchange on the least significant processor address bit has been performed, one additional exchange is required on the most significant processor address bits. By using the most significant local memory address bit for the first exchange (and all subsequent exchanges), it must be brought back into local memory again when it is needed. It is needed right after the butterfly computations have been performed on the least significant processor address bit. Using, for instance, the least significant local address bit for the exchanges instead of the most significant bit does not eliminate a need for a second communication in the most significant processor dimension. It only defers it, as shown in Figure 19.

Comparing Figures 18 and 19 we notice that

- both exchange schemes require  $n + 1$  exchanges for a processor address field with  $n$  bits.
- both exchange schemes result in a permutation of the indices
- the index permutations are different for the two exchange schemes, except after the last exchange, which results in the same final index ordering for both exchange schemes

### Data reordering

We will now consider the index permutation carefully, then determine the total communication need for the bi-section algorithm. In fact, we will show that the data reordering introduced by the bi-section algorithm is indeed an *unshuffle* operation on the initial index set. An unshuffle is the inverse of a shuffle, which perfectly interleaves the first and second half of a set of numbers. For instance, a shuffle on the set  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$  produces the set  $\{0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15\}$ .

#### Consecutive

The data reordering implied by the sequence of bi-sections can be represented formally in terms of the encoding of the address space. Let  $(\underbrace{x_{p-1}x_{p-2}\dots x_{p-n}}_{paddr}\underbrace{x_{p-n-1}x_{p-n-2}\dots x_0}_{maddr})$  be the encoding of the array index, where *paddr* denotes the processor address field and *maddr* denotes the local memory address. When the exchange is performed using the most significant local memory address bit, then the exchange in step  $k$ ,  $k = \{1, 2, \dots, n\}$ , can be described as

**WHERE**  $x_{p-k} \oplus x_{p-n-1} = 1$  **DO**  
 $(x_{p-1}x_{p-2}\dots x_{p-k+1}x_{p-k}x_{p-k-1}\dots x_{p-n}x_{p-n-1}\dots x_0) \leftarrow$   
 $\leftarrow (x_{p-1}x_{p-2}\dots x_{p-k+1}\underline{x_{p-k}}x_{p-k-1}\dots x_{p-n}\underline{x_{p-n-1}}\dots x_0)$

Proc. id	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
initial alloc.	0	4	8	12	16	20	24	28
	1	5	9	13	17	21	25	29
	2	6	10	14	18	22	26	30
	3	7	11	15	19	23	27	31
after 1st exch.	0	4	8	12	2	6	10	14
	1	5	9	13	3	7	11	15
	16	20	24	28	18	22	26	30
	17	21	25	29	19	23	27	31
after 2nd exch.	0	4	16	20	2	6	18	22
	1	5	17	21	3	7	19	23
	8	12	24	28	10	14	26	30
	9	13	25	29	11	15	27	31
after 3rd exch.	0	8	16	24	2	10	18	26
	1	9	17	25	3	11	19	27
	4	12	20	28	6	14	22	30
	5	13	21	29	7	15	23	31
after 4th exch.	0	8	16	24	4	12	20	28
	1	9	17	25	5	13	21	29
	2	10	18	26	6	14	22	30
	3	11	19	27	7	15	23	31
local comp.	0	8	16	24	4	12	20	28
	1	9	17	25	5	13	21	29
	2	10	18	26	6	14	22	30
	3	11	19	27	7	15	23	31

Figure 18: The data distribution for a bi-section based FFT with input data in normal order with consecutive data allocation. Exchanges based on the most significant local memory address bit.

Proc. id	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
initial alloc.	0	4	8	12	16	20	24	28
	1	5	9	13	17	21	25	29
	2	6	10	14	18	22	26	30
	3	7	11	15	19	23	27	31
after 1st exch.	0	4	8	12	1	5	9	13
	16	20	24	28	17	21	25	29
	2	6	10	14	3	7	11	15
	18	22	26	30	19	23	27	31
after 2nd exch.	0	4	16	20	1	5	17	21
	8	12	24	28	9	13	25	29
	2	6	18	22	3	7	19	23
	10	14	26	30	11	15	27	31
after 3rd exch.	0	8	16	24	1	9	17	25
	4	12	20	28	5	13	21	29
	2	10	18	26	3	11	19	27
	6	14	22	30	7	15	23	31
local comp.	0	8	16	24	1	9	17	25
	4	12	20	28	5	13	21	29
	2	10	18	26	3	11	19	27
	6	14	22	30	7	15	23	31
after 4th exch.	0	8	16	24	4	12	20	28
	1	9	17	25	5	13	21	29
	2	10	18	26	6	14	22	30
	3	11	19	27	7	15	23	31

Figure 19: The data distribution for a bi-section based FFT with input data in normal order with consecutive data allocation. Exchanges based on the least significant local memory address bit.

The underlined indices are the indices being exchanged. | is used to denote the separation of the node address and local memory address fields. The exchange operation can formally be described as an exchange of bits in the address. Thus, the first exchange corresponds to the exchange

$$(x_{p-1}x_{p-2} \dots x_{p-n} | x_{p-n-1}x_{p-n-2} \dots x_0) \leftarrow (\underline{x_{p-n-1}}x_{p-2} \dots x_{p-n} | \underline{x_{p-1}}x_{p-n-2} \dots x_0)$$

The correctness of this notation can be verified as follows. If we compare two array elements allocated half of the local memory apart, all address bits are the same, except bit  $x_{p-n-1}$ . In the initial consecutive allocation this bit corresponds to an index difference by  $2^{p-n-1}$ . However, with this bit in position  $p-1$ , the index difference is now  $2^{p-1}$ . It is easily verified in Figure 18 that elements in local memory that are half the local array apart differ by  $2^{p-1} = \frac{P}{2}$ .

$$(x_{p-n-1}x_{p-2} \dots x_{p-n} | x_{p-1}x_{p-n-2} \dots x_0)$$

gives the mapping from indices to addresses.

The second exchange corresponds to

$$(x_{p-n-1}x_{p-2} \dots x_{p-n} | x_{p-1}x_{p-n-2} \dots x_0) \leftarrow (x_{p-n-1}\underline{x_{p-1}} \dots x_{p-n} | \underline{x_{p-2}}x_{p-n-2} \dots x_0),$$

where  $(x_{p-n-1}x_{p-1} \dots x_{p-n} | x_{p-2}x_{p-n-2} \dots x_0)$  now defines the mapping of indices to addresses. For instance, consider index  $13 = (01101)$ . This element according to the address map after the second exchange shall be in address  $(00111)$ , which is easily verified from Figure 18.

The  $n$ th exchange corresponds to

$$\begin{aligned} & (x_{p-n-1}x_{p-1}x_{p-2} \dots x_{p-n+2}x_{p-n} | x_{p-n+1}x_{p-n-2} \dots x_0) \leftarrow \\ & \leftarrow (x_{p-n-1}x_{p-1}x_{p-2} \dots x_{p-n+2}\underline{x_{p-n+1}} | \underline{x_{p-n}}x_{p-n-2} \dots x_0), \end{aligned}$$

where again  $(x_{p-n-1}x_{p-1}x_{p-2} \dots x_{p-n+2}x_{p-n+1} | x_{p-n}x_{p-n-2} \dots x_0)$  defines the map from indices to memory addresses. The final exchange in the most significant processor dimension results in the ordering

$$\begin{aligned} & (x_{p-n-1}x_{p-1}x_{p-2} \dots x_{p-n+2}x_{p-n+1} | x_{p-n}x_{p-n-2} \dots x_0) \leftarrow \\ & \leftarrow (\underline{x_{p-n}}x_{p-1}x_{p-2} \dots x_{p-n+2}x_{p-n+1} | \underline{x_{p-n-1}}x_{p-n-2} \dots x_0) \end{aligned}$$

From this expression it is clear that a right cyclic shift on the index bits corresponding to the processor address field yields the memory address upon completion of the bi-section process. We have the following result.

Proc. id	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
initial	0	1	2	3	4	5	6	7
alloc.	8	9	10	11	12	13	14	15
after 1st exch.	0	1	2	3	8	9	10	11
after 2nd exch.	0	1	4	5	8	9	12	13
after 3rd exch.	0	2	4	6	8	10	12	14

Figure 20: The data distribution for a radix-2 FFT based on bi-section with cyclic data allocation.

**Theorem 5** *The address of index  $(x_{p-1}x_{p-2} \dots x_{p-n}x_{p-n-1} \dots x_0)$  after bi-section on successively lower order processor address bit followed by a second exchange on the most significant address bit, using the same memory bit for all exchanges is  $(x_{p-n}x_{p-1}x_{p-2} \dots x_{p-n+2}x_{p-n+1}|x_{p-n-1} \dots x_0)$ . The addresses can be computed based on an unshuffle of the  $n$  most significant index bits for  $n$  processor dimensions.*

**Corollary 2** *The map of indices to memory addresses after the  $n + 1$  bi-section steps is independent of which local memory address bit is used for the exchanges.*

Note that the bi-section algorithm with repetition of the first exchange indeed gives us an algorithm for a shuffle.

Note further that with respect to the FFT, the FFT computation still bit-reverses the indices for an in-place algorithm. Thus, for the FFT computation, the final indices are the bit-reversed values of what is shown in Figures 18 and 19.

*Cyclic*

Bi-section applied to a cyclic data allocation is illustrated in Figure 20. With cyclic data allocation, the first  $p - n$  stages require no communication. In Figure 20 there is only one local stage. In the first exchange on the most significant node address bit, the first half of the nodes exchange the content of their second memory location with the content of the first memory location of the second half of the nodes. After this exchange, each node can again perform the computations for one splitting formula, this time for stage 1 of a radix-2 FFT. The exchange proceeds on successively lower nodal dimensions, just as for consecutive data allocation. Nodes with the address bit 0 for the dimension subject to exchange, exchange their second memory location, while nodes with the address bit 1 exchange their first memory location.

Note that unlike for a consecutive data allocation it suffices to perform the exchanges once only in each processor dimension.

Upon completion of the bi-section process, the bits in the nodal address together with the local memory bit used for all exchanges have been subject to a right cyclic shift.

**Theorem 6** *In computing an FFT on data distributed over  $N = 2^n$  nodes consecutively  $n + 1$  processor dimension exchanges are required, while for cyclic data allocation  $n$  dimension exchanges suffice.*

**Theorem 7** *The bi-section process for an FFT on data allocated consecutively on  $N = 2^n$  nodes introduces an unshuffle on the  $n$  most significant index bits. For data allocated cyclicly the bi-section process introduces an unshuffle on the  $n$  most significant index bits concatenated with the index bit corresponding to the local memory address bit used for the dimension exchanges.*

Using the most significant memory dimension for the exchanges implies that blocks equal to half of the local memory are exchanged. The blocks to be exchanged are assembled with stride one. But, after each exchange, the pair of data used in each butterfly computation are separated by half of local memory.

Using the least significant memory bit for each exchange results in the pair of data used in a butterfly computation being separated with a stride of one. But, blocks for the exchange are now assembled with a stride of two.

The number of memory references are the same whether the exchange is based on the least significant or most significant local memory bit, but architectural characteristics such as page faults, communications overhead, etc., may make the strategy for selecting the local memory bit important with respect to performance.

### Communication requirements

In each bi-section step, each node exchanges half of its data with another node. For the consecutive data allocation the exchange is repeated  $n + 1$  steps, while for the cyclic data allocation  $n$  exchange steps suffice. Hence the total amount of data exchanged per node is

$$\begin{array}{cc} \text{Consecutive} & \text{Cyclic} \\ (n + 1)\frac{P}{2N} & n\frac{P}{2N} \end{array}$$

The bi-section algorithm requires about half as many element exchanges as the algorithm without explicit permutation.

Note however, that the bi-section algorithm reorders the data in addition to the bit-reversal of the in-place FFT algorithm.

### 2.4.2 Multi-section

The idea of bi-section can be generalized to multi-section to support high radix FFT. As we will see, multi-section also reduces the amount of data a node must exchange with other nodes, as suggested by Gentleman and Sande [3] and proved by Hong and Kung [5].

*Consecutive allocation*

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Initial allocation															
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
Index allocation after the first permutation															
0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
16	20	24	28	17	21	25	29	18	22	26	30	19	23	27	31
32	36	40	44	33	37	41	45	34	38	42	46	35	39	43	47
48	52	56	60	49	53	57	61	50	54	58	62	51	55	59	63
Index allocation after the second permutation															
0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63
Index allocation after the last (2nd first) permutation															
0	16	32	48	4	20	36	52	8	24	40	56	12	28	44	60
1	17	33	49	5	21	37	53	9	25	41	57	13	29	45	61
2	18	34	50	6	22	38	54	10	26	42	58	14	30	46	62
3	19	35	51	7	23	39	55	11	27	43	59	15	31	47	63

Figure 21: Data orderings for a four-section algorithm on 16 nodes. Consecutive order.

We illustrate the idea of multi-sectioning by considering computing the FFT on a 64 element array allocated consecutively in normal order to 16 nodes, as shown in Figure 21. There are four data elements local to each node. Thus, four-sectioning can be used without increasing the demands on local memory and with preserved load-balance.

In the four-sectioning algorithm three new elements for every four elements in a node are exchanged with three other nodes. In the first four-sectioning step the goal is to bring into local memory elements that differ in the two most significant bits in order to perform the first two steps of a radix-2 FFT, or one step of a radix-4 FFT. Thus, nodes that differ in their two most significant bits are involved in the four-sectioning. For instance, the first four-sectioning exchanges are performed between nodes 0, 4, 8, and 12. Similarly, exchanges are performed between nodes 1, 5, 9, and 13. Considering the latter four nodes we notice that the four-sectioning defines a matrix transposition within this set of nodes, as shown in Figure 22. Hence,  $R$ -sectioning defines a matrix transposition among independent sets of  $R$  nodes. There are  $\frac{N}{R}$  such sets for each  $R$ -sectioning.

For the FFT,  $R$ -sectioning for  $R = 2^r$  is first performed on the  $r$  most significant node address bits for normal input order and consecutive data allocation. The next  $R$ -sectioning is performed on the next  $r$  most significant bits, etc. If  $n \bmod r \neq 0$  then a  $2^{n \bmod r}$ -sectioning is required to bring in the last  $n \bmod r$  node address bits into local memory. As for bi-section, it is necessary to repeat the first  $R$ -sectioning after the last node address bits have been brought into local memory, as shown in Figure 21. The remaining FFT stages are local to each node.

Using the address notation we used for bi-section for  $2^r$ -sectioning allows us to describe the

P1	P5	P9	P13
4	20	36	52
5	21	37	53
6	22	38	54
7	23	39	55

P1	P5	P9	P13
4	5	6	7
20	21	22	23
36	37	38	39
50	53	54	55

Initially After four-sect.

Figure 22: Four-sectioning on four nodes and four local elements.

sequence of permutations as follows,

Initially

$$(x_{p-1}x_{p-2} \dots x_{p-r}x_{p-r-1}x_{p-r-2} \dots x_{p-n} | x_{p-n-1}x_{p-n-2} \dots x_0)$$

1st  $R$ -sectioning:

$$(x_{p-n-1}x_{p-n-2} \dots x_{p-n-r} \underline{x_{p-n-r-1}x_{p-n-r-2} \dots x_{p-n}} | \underline{x_{p-1}x_{p-2} \dots x_{p-r}}x_{p-n-r-1} \dots x_0)$$

2nd  $R$ -sectioning:

$$(x_{p-n-1}x_{p-n-2} \dots x_{p-n-r} \underline{x_{p-n-r-1}x_{p-n-r-2} \dots x_{p-n-2r}x_{p-2r-1}x_{p-2r-2} \dots x_{p-n}} | \underline{x_{p-r-1}x_{p-r-2} \dots x_{p-2r}}x_{p-n-r-1} \dots x_0)$$

etc.

where bits subject to exchange are underlined.

For convenience, let  $d_i$  be a radix  $2^r$  digit, i.e.,  $d_i$  is encoded in  $r$  bits. Then  $i = \{0, 1, \dots, \lceil \frac{n}{r} \rceil - 1\}$ . If  $n \bmod r \neq 0$ , then at least one of the digits must have a radix less than  $r$ . We first consider the case  $n \bmod r = 0$  and let  $\frac{n}{r} = m$ . Then, the address of an index after all  $R$ -sectioning steps can be found by an  $r$ -bit (one digit) right cyclic shift on the  $n$  leading index bits. This is apparent by considering the sequence of exchanges using the radix  $r$  digit notation.

Initially:  $(d_{m-1}d_{m-2} \dots d_0 | x_{p-n-1}x_{p-n-2} \dots x_{p-n-r}x_{p-n-r-1} \dots x_0)$

1st  $R$ -sectioning:  $(\underline{x_{p-n-1}x_{p-n-2} \dots x_{p-n-r}} d_{m-1}d_{m-2} \dots d_0 | \underline{d_{m-1}} x_{p-n-r-1} \dots x_0)$

2nd  $R$ -sectioning:  $(x_{p-n-1}x_{p-n-2} \dots x_{p-n-r} \underline{d_{m-1}} d_{m-3} d_{m-4} \dots d_0 | \underline{d_{m-2}} x_{p-n-r-1} \dots x_0)$

3rd  $R$ -sectioning:  $(x_{p-n-1}x_{p-n-2} \dots x_{p-n-r} d_{m-1} \underline{d_{m-2}} d_{m-4} \dots d_0 | \underline{d_{m-3}} x_{p-n-r-1} \dots x_0)$

$m$ th  $R$ -sectioning:  $(x_{p-n-1}x_{p-n-2} \dots x_{p-n-r} d_{m-1}d_{m-2}d_{m-3} \dots \underline{d_1} | \underline{d_0} x_{p-n-r-1} \dots x_0)$

$m + 1$ th  $R$ -sectioning:  $(\underline{d_0} d_{m-1} d_{m-2} d_{m-3} \dots d_1 | \underline{x_{p-n-1}x_{p-n-2} \dots x_{p-n-r}} x_{p-n-r-1} \dots x_0)$

As a concrete example, consider eight-sectioning on a 512 node system for the computation of an FFT on 4k elements. Each node holds eight elements. The sequence of index allocations to memory addresses are

Initially:  $(11 \ 10 \ 9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ | \ 2 \ 1 \ 0)$

1st eight-sectioning:  $(2 \ 1 \ 0 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ | \ 11 \ 10 \ 9)$

2nd eight-sectioning:  $(2 \ 1 \ 0 \ 11 \ 10 \ 9 \ 5 \ 4 \ 3 \ | \ 8 \ 7 \ 6)$

3rd eight-sectioning:  $(2 \ 1 \ 0 \ 11 \ 10 \ 9 \ 8 \ 7 \ 6 \ | \ 5 \ 4 \ 3)$

4th eight-sectioning:  $(5 \ 4 \ 3 \ 11 \ 10 \ 9 \ 8 \ 7 \ 6 \ | \ 2 \ 1 \ 0)$

where we only show the indices of the bits.

Now, we will also consider the case where  $n \bmod r \neq 0$  through an example. Let a 1024 point array be allocated to 128 nodes with eight elements per node. Two stages of eight-section is possible, followed by one step of bi-section. Then, the first eight-sectioning must be repeated.

Initially:  $(9 \ 8 \ 7 \ 6 \ 5 \ 4 \ 3 \ | \ 2 \ 1 \ 0)$

1st eight-sectioning:  $(2 \ 1 \ 0 \ 6 \ 5 \ 4 \ 3 \ | \ 9 \ 8 \ 7)$

2nd eight-sectioning:  $(2 \ 1 \ 0 \ 9 \ 8 \ 7 \ 3 \ | \ 6 \ 5 \ 4)$

Bi-sectioning:  $(2 \ 1 \ 0 \ 9 \ 8 \ 7 \ 6 \ | \ 3 \ 5 \ 4)$

Local perm.:  $(2 \ 1 \ 0 \ 9 \ 8 \ 7 \ 6 \ | \ 5 \ 4 \ 3)$

Last eight-sectioning:  $(5 \ 4 \ 3 \ 9 \ 8 \ 7 \ 6 \ | \ 2 \ 1 \ 0)$

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120
1	9	17	25	33	41	49	57	65	73	81	89	97	105	113	121
2	10	18	26	34	42	50	58	66	74	82	90	98	106	114	122
3	11	19	27	35	43	51	59	67	75	83	91	99	107	115	123
4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124
5	13	21	29	37	45	53	61	69	77	85	93	101	109	117	125
6	14	22	30	38	46	54	62	70	78	86	94	102	110	118	126
7	15	23	31	39	47	55	63	71	79	87	95	103	111	119	127

Figure 23: Initial allocation

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15
16	24	17	25	18	26	19	27	20	28	21	29	22	30	23	31
32	40	33	41	34	42	35	43	36	44	37	45	38	46	39	47
48	56	49	57	50	58	51	59	52	60	53	61	54	62	55	63
64	72	65	73	66	74	67	75	68	76	69	77	70	78	71	79
80	88	81	89	82	90	83	91	84	92	85	93	86	94	87	95
96	104	97	105	98	106	99	107	100	108	101	109	102	110	103	111
112	120	113	121	114	122	105	123	116	124	117	125	118	126	119	127

Figure 24: Index allocation after the first 8–section.

Note that by performing the local (shuffle) permutation, the end result is again an  $r$ –bit right cyclic shift of the  $n$  leading index bits. Figures 23 through 27 give an example in which 128 elements are allocated to 16 nodes and one eight–sectioning is made followed by one bi–sectioning.

Note that when there are more local data elements than what corresponds to  $R$  for the  $R$ –sectioning, then blocks of  $\frac{P}{NR}$  elements are exchanged between pairs of nodes, instead of single elements.

### *Cyclic Allocation*

We have so far only considered consecutive data allocation. Figure 28 illustrates four–sectioning for 64 data elements allocated cyclicly in normal order to 16 nodes ( $p = 6, n = 4$ ). The first four–sectioning step is a matrix transposition within subsets of four nodes, defined by the two highest order nodal address bits, just as for consecutive data allocation. The number of such subsets are defined by the remaining nodal address bits. The final index allocation depends upon which local memory bits are used for the exchanges, just as for consecutive data allocation.

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
0	64	1	65	2	66	3	67	4	68	5	69	6	70	7	71
16	80	17	81	18	82	19	83	20	84	21	85	22	86	23	87
32	96	33	97	34	98	35	99	36	100	37	101	38	102	39	103
48	112	49	113	50	114	51	115	52	116	53	117	54	118	55	119
8	72	9	73	10	74	11	75	12	76	13	77	14	78	15	79
24	88	25	89	26	90	27	91	28	92	29	93	30	94	31	95
40	104	41	105	42	106	43	107	44	108	45	109	46	110	47	111
56	120	57	121	58	122	59	123	60	124	61	125	62	126	63	127

Figure 25: Index allocation after the bi-section exchange.

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
0	64	1	65	2	66	3	67	4	68	5	69	6	70	7	71
8	72	9	73	10	74	11	75	12	76	13	77	14	78	15	79
16	80	17	81	18	82	19	83	20	84	21	85	22	86	23	87
24	88	25	89	26	90	27	91	28	92	29	93	30	94	31	95
32	96	33	97	34	98	35	99	36	100	37	101	38	102	39	103
40	104	41	105	42	106	43	107	44	108	45	109	46	110	47	111
48	112	49	113	50	114	51	115	52	116	53	117	54	118	55	119
56	120	57	121	58	122	59	123	60	124	61	125	62	126	63	127

Figure 26: Index allocation after the local shuffle.

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
0	64	8	72	16	80	24	88	32	96	40	104	48	112	56	120
1	65	9	73	17	81	25	89	33	97	41	105	49	113	57	121
2	66	10	74	18	82	26	90	34	98	42	106	50	114	58	122
3	67	11	75	19	83	27	91	35	99	43	107	51	115	59	123
4	68	12	76	20	84	28	92	36	100	44	108	52	116	60	124
5	69	13	77	21	85	29	93	37	101	45	109	53	117	61	125
6	70	14	78	22	86	30	94	38	102	46	110	54	118	62	126
7	71	15	79	23	87	31	95	39	103	47	111	55	119	63	127

Figure 27: Index allocation after the last exchange.



Allocation	$\frac{P}{N} \geq N$	$\frac{P}{N} < N$
Consecutive	$(N - 1)\frac{P}{N^2}2$	$(\frac{P}{N} - 1)(\lfloor \frac{n}{p-n} \rfloor + 1) + (2^{n \bmod p-n} - 1)\frac{P}{N2^{n \bmod p-n}}$
Cyclic	$(N - 1)\frac{P}{N^2}$	$(\frac{P}{N} - 1)\lfloor \frac{n}{p-n} \rfloor + (2^{n \bmod p-n} - 1)\frac{P}{N2^{n \bmod p-n}}$

Table 6: The number of elements exchanged per node for optimal multi-sectioning.

Cyclic data allocation:  $(2^r - 1)\frac{P}{N2^r} \lfloor \frac{n}{r} \rfloor + (2^{n \bmod r} - 1)\frac{P}{N2^{n \bmod r}}$

Consecutive data allocation:  $(2^r - 1)\frac{P}{N2^r} (\lfloor \frac{n}{r} \rfloor + 1) + (2^{n \bmod r} - 1)\frac{P}{N2^{n \bmod r}}$

Minimizing the communication requirements is equivalent to maximizing  $r$ . Conserving memory and balancing the load requires that  $2^r \leq \frac{P}{N}$ , i.e.,  $r \leq p - n$ . Furthermore, clearly  $r \leq n$ , since the  $R$ -sectioning will at most include all nodal dimensions. Thus, we have the communication requirements for optimal  $R$ -sectioning as give in Table 6.

**Comparing of bi-section and multi-section**

Multi-sectioning offers approximately a factor of  $\frac{2^r-1}{2 \cdot 2^r}r$  reduction in the required data motion over the bi-section algorithm. This rather large difference does not however automatically translate into a corresponding advantage in the communication time, as we will see in discussing multi-sectioning on binary cubes.

Bi-sectioning and multi-sectioning results in a permutation of the indices, in addition to the bit-reversal inherent in in-place FFT algorithms. By carefully managing the exchanges and the local memory address bits used for the exchanges, the final index allocation corresponds to a shuffle on the index bits corresponding to node addresses repeated  $r$  times for consecutive data allocation, while for cyclic data allocation the shuffle permutation includes local memory address bits. An ordered FFT, i.e., a transform that has the same input and output orderings requires an unshuffle combined with a bit-reversal. We will discuss reorderings further below.

In all derivations above the input order was normal. With the input in bit-reversed order the traversal of the address bits proceeds from the lowest to the highest order bit. With respect to the communication issues the roles of the consecutive and cyclic mapping are interchanged. The nodal communication requirements are summarized in Table 7.

The communication needs for the algorithm without explicit permutations is the same for both consecutive and cyclic data allocation. A bi-section algorithm has a factor two lower communication needs per node for cyclic data allocation and normal input order. For a consecutive data allocation and normal input order, one extra communication is needed. For a cyclic data allocation and normal input order, multi-sectioning offers a potential for a  $\log_2 N$  factor reduction in the communication requirements, compared to the algorithm without explicit data reordering. For the consecutive data allocation the reduction in data motion requirements may be reduced to  $\frac{1}{2} \log_2 N$ .

Input Order	Algorithm	Consecutive	Cyclic
Normal	No expl. perm.	$\frac{P}{N} \log_2 N$	$\frac{P}{N} \log_2 N$
	Bi-section	$\frac{P}{2N} (\log_2 N + 1)$	$\frac{P}{2N} \log_2 N$
	Multi-section	$\sim \frac{P}{N} (\lceil \frac{\log_2 N}{\log_2 \frac{P}{N}} \rceil + 1)$	$\sim \frac{P}{N} \lceil \frac{\log_2 N}{\log_2 \frac{P}{N}} \rceil$
Bit-reversed	No expl. perm.	$\frac{P}{N} \log_2 N$	$\frac{P}{N} \log_2 N$
	Bi-section	$\frac{P}{2N} \log_2 N$	$\frac{P}{2N} (\log_2 N + 1)$
	Multi-section	$\sim \frac{P}{N} \lceil \frac{\log_2 N}{\log_2 \frac{P}{N}} \rceil$	$\sim \frac{P}{N} (\lceil \frac{\log_2 N}{\log_2 \frac{P}{N}} \rceil + 1)$

Table 7: Communication requirements per node for a few FFT algorithms.

For a node limited communication system, i.e., a communication system in which the total transfer rate into or out of a node is the limiting factor, the total number of element transfers is a good measure of the required communication time. This simple model is a good approximation of the communication system on the Intel series of machines, the Ncube, and the CM-5. However, in some communication systems, like the Connection Machine CM-2 system, the bandwidth per communications channel is the limiting factor. The bandwidth per node grows in proportion to the number of channels being used. For such systems it is necessary to consider how the network characteristics can be exploited before a final comparison of the communication times for the different algorithms can be made.

## 2.5 Twiddle factor allocation

In this section we will consider the memory requirements for twiddle factors, when precomputation is desired. With precomputed twiddle factors it is of particular importance to either ensure reuse of twiddle factors for different FFT stages, as in the uniprocessor case, or at least to make sure that the number of needed twiddle factors per stage is reduced for *every* node. In a worst case scenario,  $\frac{P}{N}$  twiddles may be required for most stages in at least one node. We will first consider the twiddle factor use in the nodes for the FFT algorithm without explicit permutations, then consider the multi-sectioning algorithm.

### 2.5.1 Twiddle factor storage using no explicit permutations

#### Radix-2 DIF FFT

For an *in-place* radix-2 DIF FFT, the twiddle factors required for the butterfly computation in stage  $q$  is  $\omega_P^{(j_{p-q-1}) \times (j_{p-q-2} j_{p-q-3} \dots j_0) 2^q}$ . The exponent of the twiddle factor for a pair of complex elements in a butterfly computation is formed from address bits 0 through  $p - q - 2$  shifted left  $q$  steps with an end-off shift.

With a *consecutive* data allocation at least one node needs  $\frac{P}{N}$  twiddle factors for each of the first  $n - 1$  stages. The sets of twiddle factors for different stages are disjoint. For instance, consider the node with address  $(j_{p-1} j_{p-2} \dots j_{p-n+1} j_{p-n}) = (11 \dots 10)$ . This node contains the

Stage 0 (1110 xxx)	Stage 1 (110x xx0)	Stage 2 (10xx x00)	Stage 3 (0xxx 000)	Stage 4 (xxx0 000)	Stage 5 (xx00 000)	Stage 6 (x000 000)
112	96	64	0	0	0	0
113	98	68	8	16	32	64
114	100	72	16	32	64	
115	102	76	24	48	96	
116	104	80	32	64		
117	106	84	40	80		
118	108	88	48	96		
119	110	92	56	112		

Table 8: Twiddle exponents in node 30 of a 32 node system for a DIF FFT on normal order input, consecutive data allocation.

data indices  $(11 \dots 10 | \{j_{p-n-1} j_{p-n-2} \dots j_0\})$ , with the consecutive allocation. Braces are used to denote all permutations of the indices within the braces. Shifting this set of addresses left by one step yields a new set of addresses if  $n > 2$ . This property holds for  $n - 1$  left shifts, i.e., the first  $n - 1$  stages. For  $n = 1$  it is easily seen that  $\frac{P}{N}$  twiddle factors are needed, and for  $n = 2$  node (11) requires  $3\frac{P}{2N} - 1$  twiddles. Hence, at least  $(n - 1)\frac{P}{N}$  twiddles are required in at least one node.

An example of twiddle exponents for a 32 node system with consecutive, normal order data allocation is given in Table 8. Note that no sharing of twiddles occurs in the first  $n - 1$  stages. The remaining stages do not include any node address bits in computing twiddle exponents. Successive local stages use subsets of twiddles from the preceding stage.

For a *cyclic* allocation the first  $n$  stages are local to a node. The sets of twiddle factor indices required for the stages local to a node are

$$(\{j_{p-2} j_{p-3} \dots j_n\} | j_{n-1} \dots j_0), (\{j_{p-3} j_{p-4} \dots j_n\} | j_{n-1} \dots j_0)2, \dots, (n-1 \dots j_0)2^{p-n-1},$$

for stages 0, 1, etc., or a maximum total of  $\sum_{m=1}^{p-n} 2^{p-n-m} = \frac{P}{N} - 1$  for any node.

After the first  $p - n$  stages, it follows from the derivation of the DIF FFT that the remaining  $n$  stages consist of  $\frac{P}{N}$  independent FFTs of size  $N$ , each with one element per node. All  $\frac{P}{N}$  FFTs have the same twiddle factor for a given butterfly stage and node. A maximum of  $n - 1$  twiddle factors per node is needed for the inter-node communication stages (one for each butterfly stage, except the last). Hence, for cyclic data allocation and a radix-2 DIF FFT of size  $2^p$  computed on  $N$  nodes,  $n < p$ , the maximum number of distinct twiddle factors needed in a node is  $\frac{P}{N} + n - 2$ .

Figure 29 shows an example of the twiddle factor exponents need for the different data elements and the different stages for a 64 point DIFT FFT on an eight node system with normal order cyclic allocation. Notice that node seven needs  $4+2+1$  different twiddle factors, i.e.,  $\frac{P}{N} - 1$  different twiddle factors. Notice also that stage 0, 1 and 2 are local, while stage 3, 4 and 5 require communication. For the latter stages the twiddle factor exponents are the same for all

local elements.

### Radix–2 DIT FFT

For a radix–2 DIT FFT algorithm without explicit permutations, the exponents of the twiddle factors for stage  $q$  are  $\omega_P^{(j_{p-q-1})(j_{p-q}j_{p-q+1}\dots j_{p-1})2^{p-1-q}}$ ,  $q \in [0, p - 1]$ , and  $q = 0$  for the first stage. Note that the address is bit–reversed and shifted for the proper exponent. The twiddle factors for stage 0 are all  $\omega_P^0$ . If the FFT of size  $P = 2^p$  is computed on  $P$  nodes, then node  $P - 1$  requires  $p - 1$  distinct twiddle factors. For a  $P$  element transform performed on  $N$  nodes with normal order data allocated consecutively, stages 1 through  $n - 1$  each requires one twiddle factor per stage and node. All  $\frac{P}{N}$  local elements have the same twiddle factor in a given node. The last  $p - n$  stages are local, and the maximum total number of twiddle factors required per node is  $\frac{P}{N} - 1$ .

We will now prove the claims made above. In the consecutive data allocation the assignment of data indices to nodes is  $(j_{p-1}j_{p-2}\dots j_{p-n}|\{j_{p-n-1}j_{p-n-2}\dots j_0\})$ . Clearly, for  $q < n$  none of the data index bits mapped into local memory enters into the twiddle factor exponent. Hence, all local data elements have the same twiddle factor for the stages requiring inter–node communication. The last stage, which is local, requires  $\frac{P}{2N}$  twiddle factors, since the set of exponents are computed from  $(\{j_0\}(j_1j_2\dots j_{p-n-1})|j_{p-n}\dots j_{p-1})$ . In node zero the twiddle factors for stages  $n < q < p - 1$  are always a subset of the twiddle factors for stage  $p - 1$ , whereas in node  $N - 1$  successive stages have unique twiddle factors. Hence, a maximum total of  $\sum_{k=1}^{p-n} \frac{P}{2^k N} = \frac{P}{N} - 1$  different twiddle factors per node is needed for the local stages, and the second part of the claim has been verified. The total number of twiddle factors in a node for normal order input and consecutive data allocation is  $\frac{P}{N} + n - 2$  for the radix–2 DIT FFT.

With *cyclic* allocation the  $p - n$  most significant bits are mapped into local memory. The last  $n$  steps require communication, and different local elements often have different twiddle factors. For instance, consider node  $(011\dots 1)$ , which in the cyclic allocation is assigned indices  $(\{j_{p-1}j_{p-2}\dots j_n\}|011\dots 1)$ . The last stage requires twiddle factors with indices  $(1\dots 110|\{j_nj_{n+1}\dots j_{p-1}\})$ , or  $\frac{P}{N}$  twiddle factors. The second to last stage requires another  $\frac{P}{N}$  twiddle factors in node  $(011\dots 1)$ , since the index set  $(1\dots 110|\{j_n|j_{n+1}\dots j_{p-1}\}0)$  is disjoint from the set for the last stage. In general, at least  $(n - 1)\frac{P}{N}$  twiddle factors are required in a node for normal order input and cyclic data allocation for the radix–2 DIT FFT.

### Bit–reversed input

With the input in bit–reversed order, the traversal of the bits in the address field is from the least significant bit to the most significant bit. The indices used for twiddle factor computation for the DIF FFT are obtained by using bit–reversed addresses, instead of addresses in normal order. Analogously, the DIT FFT uses addresses in normal order for the index computation, instead of bit–reversed addresses for normal order input.

With the input data in bit–reversed order the DIF FFT requires the least twiddle factor storage for the consecutive data allocation, while the DIT FFT requires the least storage for the cyclic data allocation. The preferred combinations of data allocation and FFT type are the opposite to those preferred with normal order input.

### Reduced twiddle factor storage

P0	P1	P2	P3	P4	P5	P6	P7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Initial index allocation

P0	P1	P2	P3	P4	P5	P6	P7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Twiddle exponents for stage 0.

P0	P1	P2	P3	P4	P5	P6	P7
0	2	4	6	8	10	12	14
16	18	20	22	24	26	28	30
0	2	4	6	8	10	12	14
16	18	20	22	24	26	28	30

Twiddle exponents for stage 1.

P0	P1	P2	P3	P4	P5	P6	P7
0	4	8	12	16	20	24	28
0	4	8	12	16	20	24	28
0	4	8	12	16	20	24	28
0	4	8	12	16	20	24	28

Twiddle exponents for stage 2.

P0	P1	P2	P3	P4	P5	P6	P7
				0	8	16	24
				0	8	16	24
				0	8	16	24
				0	8	16	24
				0	8	16	24
				0	8	16	24
				0	8	16	24
				0	8	16	24

Twiddle exponents for stage 3.

P0	P1	P2	P3	P4	P5	P6	P7
		0	16			0	16
		0	16			0	16
		0	16			0	16
		0	16			0	16
		0	16			0	16
		0	16			0	16
		0	16			0	16
		0	16			0	16

Twiddle exponents for stage 4.

Figure 29: Twiddle factor exponents for DIF FFT and cyclic data allocation

For the consecutive data allocation, normal order input, and a radix-2 DIT FFT, the set of twiddle factor indices in the last stage is  $(\{j_1 j_2 \dots j_{n-1}\} | j_n \dots j_{p-1})$ . The highest order bit  $j_1$  corresponds to bit position  $p-2$ . Hence,  $(\{1 j_2 \dots j_{n-1}\} | j_n \dots j_{p-1}) = \frac{P}{4} + (\{0 j_2 \dots j_{n-1}\} | j_n \dots j_{p-1})$ . But,  $\omega_P^{j+\frac{P}{4}} = \omega_P^j \cdot e^{-\frac{2\pi i}{P} \frac{P}{4}} = -i \cdot \omega_P^j$ . Multiplication by  $-i$ , which implies a 90-degree rotation, simply involves exchanging the real and imaginary parts of  $\omega_P^j$  and negating the imaginary part. Therefore, half of the twiddle factors can be obtained from the other half with no arithmetic. This property is true for all local stages. The same property is true for

- decimation-in-frequency FFT, cyclic data allocation, and normal input order,
- decimation-in-time FFT, cyclic data allocation, and bit-reversed input order,
- decimation-in-frequency FFT, consecutive data allocation, and bit-reversed input order.

The observation can be generalized to bits  $p-3, p-4, \dots$  in the twiddle factor index, but complex arithmetic is required for rotations of less than 45-degrees. For bit  $p-3$ , which is associated with a 45-degree rotation,  $\omega_P^{\frac{P}{8}} = -\frac{1+i}{\sqrt{2}}$ , which at first seem to require complex multiplication. However, the reflective symmetry between sine and cosine can be used to avoid this multiplication by using say  $\cosine(\frac{\pi}{2} - \alpha)$  instead of  $\sine(\alpha)$ .

### Summary of twiddle storage for radix-2 DIF and DIT FFT

For radix-2 FFT without explicit permutation of the data, we conclude that the preferred combinations of data allocation, input order, and FFT type with respect to twiddle factor storage are:

- normal input order, consecutive data allocation, decimation-in-time FFT
- normal input order, cyclic data allocation, decimation-in-frequency FFT
- bit-reversed input order, consecutive data allocation, decimation-in-frequency FFT
- bit-reversed input order, cyclic data allocation, decimation-in-time FFT

The storage requirements and the formula for the twiddle factor index computations are summarized in Table 9. With 90-degree rotations performed “on-the-fly”, the storage requirements for local twiddles is reduced by a factor of two compared to what is stated in Table 9.

Allocating twiddle factor storage uniformly across all nodes yields a total twiddle factor storage of  $P + (n-2)N$ , which for  $P \gg N$  is about twice the storage required on a sequential or shared memory computer. For  $P = N$ , uniform twiddle factor storage across nodes yields a total storage of  $(n-1)N$ , which exceeds the sequential storage by a factor of approximately  $2(n-1)$ .

### Radix- $R$ twiddle factor storage for algorithms without explicit permutations

The relationship between consecutive and cyclic allocation and normal and bit-reversed input order is independent of the radix of the algorithm. Thus, we only consider the the twiddle factor arrangement for the combinations requiring the fewest twiddle factors.

FFT	Data alloc.	Twiddle index stage $q$	Max. number of twiddles per node
Normal input order			
DIT	consec.	$\{j_{p-q}j_{p-q+1} \dots j_{p-n-1}\}   j_{p-n} \dots j_{p-1} 2^{p-1-q}$	$\frac{P}{N} + n - 2$
	cyclic	$j_{p-q}j_{p-q+1} \dots j_{n-1}   \{j_n \dots j_{p-1}\} 2^{p-1-q}$	$\geq (n-1) \frac{P}{N}$
DIF	consec.	$j_{p-q-2}j_{p-q-3} \dots j_{p-n}   \{j_{p-n-1} \dots j_0\} 2^q$	$\geq (n-1) \frac{P}{N}$
	cyclic	$\{j_{p-q-2}j_{p-q-3} \dots j_n\}   j_{n-1} \dots j_0 2^q$	$\frac{P}{N} + n - 2$
Bit-reverse input order			
DIT	consec.	$j_{q-1}j_{q-2} \dots j_{p-n}   \{j_{p-n-1} \dots j_0\} 2^{p-1-q}$	$\geq (n-1) \frac{P}{N}$
	cyclic	$\{j_{q-1}j_{q-2} \dots j_n\}   j_{n-1} \dots j_0 2^{p-1-q}$	$\frac{P}{N} + n - 2$
DIF	consec.	$j_{q+1}j_{q+2} \dots j_{n-1}   \{j_n \dots j_{p-1}\} 2^q$	$\frac{P}{N} + n - 2$
	cyclic	$\{j_{q+1}j_{q+2} \dots j_{p-n-1}\}   j_{p-n} \dots j_{p-1} 2^q$	$\geq (n-1) \frac{P}{N}$

Table 9: Radix-2 twiddle factor storage for DIF and DIT FFT for different data allocation schemes and input orders.

Recall that for a DIF FFT, the twiddle factor index for data in location  $(d_{u-1}d_{u-2} \dots d_0)$  is  $\widehat{d_{u-1}} \times (d_{u-2}d_{u-3} \dots d_0)$  for the first stage. For the second radix- $2^r$  stage the set of twiddle factor indices are  $\widehat{d_{u-2}} \times (d_{u-3}d_{u-4} \dots d_0)2^r$ . In general, for a radix- $2^r$  *in-place* DIF FFT on normal order input data, the twiddle factor index for the data in location  $(d_{u-1}d_{u-2} \dots d_0)$  after the  $s$ th radix- $2^r$  stage is  $\widehat{d_{u-s-1}} \times (d_{u-s-2}d_{u-s-3} \dots d_0)2^{sr}$ .

For simplicity, we assume that  $u$  and  $n$  are multiples of  $r$ . The twiddle factor indices for stage  $s$  required in node  $(d_{\frac{n}{r}-1} \dots d_0)$  are  $\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2}d_{u-s-3} \dots d_{\frac{n}{r}}\} d_{\frac{n}{r}-1} \dots d_0)2^{sr}$ . When  $\frac{P}{N}$  is a multiple of  $2^r$ , then  $(\frac{P}{N} - 1)$  twiddle factors are needed for the local stages.

The stages requiring communication correspond to computing  $\frac{P}{N}$  independent FFTs of size  $N$ , each with one element per node. All  $\frac{P}{N}$  FFTs require the same set of twiddle factors in a node. A total of at most  $(\lceil \frac{n}{r} \rceil - 1)(2^r - 1)$  twiddle factors are needed in a node for these stages (one set for each radix- $2^r$  butterfly stage, except the last stage).

To summarize, the maximum number of distinct twiddle factors needed in a node is  $\frac{P}{N} + (\lceil \frac{n}{r} \rceil - 1)(2^r - 1) - 1$  for cyclic data allocation, normal input order, and a radix- $2^r$  DIF FFT of size  $P$  computed on  $N$  nodes,  $N \leq P$ . Allocating twiddle factor storage uniformly across all nodes yield a total twiddle factor storage of  $P - N + (\lceil \frac{n}{r} \rceil - 1)(2^r - 1)N$ , which for  $P \gg N$  is about twice the storage required on a shared memory computer. The same twiddle storage is required for a bit-reversed input order, and a consecutive data allocation. Normal input order and consecutive data allocation, or cyclic allocation with bit-reversed input order would require considerably more storage, for the same reasons as in the radix-2 case [13].

For a high radix DIT FFT, the indices of the twiddle factors are all zero for the first stage,  $j \times \widehat{d_{u-1}} 2^{p-2r}$  for the second radix- $2^r$  stage, and  $j \times (\widehat{d_{u-s}} \dots \widehat{d_{u-1}}) 2^{p-(s+1)r}$  for stage number  $s$ . Note, that the address is bit-reversed and shifted for the proper exponent. If the  $P$  complex data points are allocated consecutively and are in normal order, then the data in address location

FFT	Data alloc.	Twiddle index stage $s$	Max. number of twiddles per nodes
Normal input order			
DIT	consec.	$\{j\} \times (\{\widehat{d_{u-s}} \dots \widehat{d_{u-\frac{n}{r}-1}}\} \widehat{d_{u-\frac{n}{r}}} \dots \widehat{d_{u-1}}) 2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2} d_{u-s-3} \dots d_{\frac{n}{r}}\} d_{\frac{n}{r}-1} \dots d_0) 2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$
Bit-reversed input order			
DIT	cyclic.	$\{\widehat{d_{u-s-1}}\} \times (\{d_{u-s-2} d_{u-s-3} \dots d_{\frac{n}{r}}\} d_{\frac{n}{r}-1} \dots d_0) 2^{sr}$	$\frac{P}{N} + \frac{n}{r} - 2$
DIF	consec.	$\{j\} \times (\{\widehat{d_{u-s}} \dots \widehat{d_{u-\frac{n}{r}-1}}\} \widehat{d_{u-\frac{n}{r}}} \dots \widehat{d_{u-1}}) 2^{p-(s+1)r}$	$\frac{P}{N} + \frac{n}{r} - 2$

Table 10: Radix- $2^r$  twiddle factor storage as a function of input order.

$(d_{u-1} d_{u-2} \dots d_0)$  requires twiddle factors with indices  $\{j\} \times (\{\widehat{d_{u-s}} \dots \widehat{d_{u-\frac{n}{r}-1}}\} \widehat{d_{u-\frac{n}{r}}} \dots \widehat{d_{u-1}}) 2^{p-(s+1)r}$  for stage  $s$  of an in-place DIT algorithm. With a consecutive data allocation the node address bits form the high order bits of the element index. The first  $\frac{n}{r}$  radix- $2^r$  butterfly stages correspond to  $\frac{P}{N}$  independent FFTs of size  $N$ . All these FFTs require the same set of twiddle factors. The local addresses do not enter into the index computation. Moreover, the first stage does not require any twiddle factor. The last  $u - \frac{n}{r}$  radix- $2^r$  stages are local to a node. The maximum total number of twiddle factors required in a node is  $\frac{P}{N} + (\lceil \frac{n}{r} \rceil - 1)(2^r - 1) - 1$ , the same as for cyclic data allocation, normal input order, and in-place DIF FFT. The set of twiddle factors required in a node is identical to those required for consecutive data allocation, bit-reversed input order and a DIF, in-place FFT. The number of twiddle factors required for a DIT FFT with input data in bit-reversed order and a consecutive data allocation is excessive, see [13].

The preferred combinations of data allocation and FFT type for radix- $R$  FFT are summarized in Table 10.

The Inverse Discrete Fourier Transform can be computed as a Discrete Fourier Transform by using conjugate twiddle factors.

**Remark:** It is important to notice that in a distributed FFT, when both forward and inverse FFT are required that a forward DIT on data in normal order and an inverse DIF FFT on data in bit-reversed order can use the same twiddle tables. This combination of FFT is optimal for consecutive data allocation. For bit-reversed input data a forward DIF FFT and an inverse DIT FFT yields the minimum twiddle storage, and again, the same twiddle table can be used for both transforms (with conjugation).

### 2.5.2 Twiddle factor storage with explicit permutations

We start with considering the computation of an FFT on 64 elements distributed evenly over 16 nodes in normal order with consecutive data allocation. Using multi-sectioning, the index allocation for four-sectioning was given in Figure 21. The twiddle exponents needed for the

different radix-2 stages are shown in Figure 2.5.2.

The following observations can be made for a DIT FFT:

- After the first  $2^r$ -sectioning, all nodes use the same twiddles for the first  $r$  radix-2 stages. The replication of twiddles is  $N$ -fold.
- As the  $2^r$  sectioning proceeds, the replication of twiddle factors is reduced by a factor of  $2^r$  for each  $2^r$ -sectioning stage. The number of different sets of twiddle factors is  $2^{(s-1)r}$  after the  $s$ th  $2^r$ -sectioning step,  $s = 1, 2, \dots, \frac{n}{r}$ . After the last  $2^r$ -sectioning, the number of different sets of twiddle factors is  $2^{n-r}$ . Each set of twiddle factors is of size  $2^{r-1}$ .
- After each  $2^r$ -sectioning, the twiddles for the first  $r - 1$  stages are subsets of the twiddles for the  $r$ th stage. However, the twiddles required after a  $2^r$ -sectioning may be entirely different from any preceding stage.
- Using 90-degree rotations to reduce twiddle factor storage requires that the preceding bit is local when more than one local twiddle is required for a stage. This requirement puts constraints on how the last exchange for the processor address field is performed, if less than  $r$  bits are involved, i.e., when  $n \bmod r \neq 0$ .

The twiddle exponents in stage  $q$  of a radix-2 DIT FFT is  $(i_{p-q-1}) \cdot (i_{p-q}i_{p-q+1} \dots i_{p-1})2^{p-q-1}$ . We assume that the FFT stages are numbered from 0 to  $p - 1$ . Table 13 shows the twiddle indices for the first few stages. For the permutation on the  $n \bmod r$  least significant node address bits, it is also true that the twiddles required for radix-2 stages  $1 - (n \bmod r - 1)$  form a subset of the twiddles needed for stage  $n \bmod r$ . We use the example provided earlier for computing a 128 element FFT on 16 nodes to illustrate this point, Figures 11 and 12.

Assume that bits  $p - 1$  to  $p - q$  have been processed, and are part of the processor address field after the multi-sectioning, and that bits  $p - q - 1$  through  $p - q - r$  are available locally after the multi-section stage. Then, the number of twiddles required locally for radix-2 stages  $q$  through  $q + r - 1$  are given in Table 14. This Table also gives the twiddle exponents, and the index of the data for which the twiddles are used. The set of indices present within a processor's memory at a given time is included within braces  $\{\dots\}$ .

Note that the leading bit in  $(i_{p-q-1}i_{p-q} \dots i_{p-1})2^{p-q-1}$  always is in location  $p - 2$ . Thus,  $(1i_{p-q} \dots i_{p-1})2^{p-q-1} = (0i_{p-q} \dots i_{p-1})2^{p-q-1} + \frac{P}{4}$ . This property implies that for the second and all following local stages after a multi-section step, the size of the twiddle factor table can be reduced by a factor of two by performing 90-degree rotations on-the-fly, just as in an FFT [12] without explicit reordering.

The number of twiddle factors required after each  $2^r$ -section step on  $r$  bits is  $1, 2, \dots, 2^{r-1}$  for the first, second, third, etc. radix-2 stage after the  $2^r$ -section. Using 90-degree rotations, the required number of twiddles for successive radix-2 stages after each  $2^r$ -section step is  $1, 1, 2, \dots, 2^{r-2}$ . For each radix-2 butterfly stage after a  $2^r$ -sectioning step, the twiddles form a subset of the twiddles required in the last radix-2 stage before then next  $2^r$ -sectioning. Hence, the number of twiddle factors required in a node for each  $2^r$ -sectioning stage is  $2^{r-1}$  and  $2^{r-2} + 1$ , respectively. The total twiddle factor storage is:

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Initial allocation															
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63
Twiddle exponents in first radix-2 stage.															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Twiddle exponents in second radix-2 stage.															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
Twiddle exponents in third radix-2 stage.															
0	16	8	24	0	16	8	24	0	16	8	24	0	16	8	24
0	16	8	24	0	16	8	24	0	16	8	24	0	16	8	24
Twiddle exponents in fourth radix-2 stage.															
0	8	4	12	0	8	4	12	0	8	4	12	0	8	4	12
16	24	20	28	16	24	20	28	16	24	20	28	16	24	20	28
Twiddle exponents in fifth radix-2 stage.															
0	8	2	6	16	20	18	22	8	12	10	14	24	28	26	30
0	8	2	6	16	20	18	22	8	12	10	14	24	28	26	30
Twiddle exponents in sixth radix-2 stage.															
0	2	1	3	8	10	9	11	4	6	5	7	12	14	13	15
16	18	17	19	24	26	25	27	20	22	21	23	28	30	29	31

Figure 30: Radix-2 twiddle exponents for normal order input, consecutive allocation.

Stage	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48
4	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8
	32	40	32	40	32	40	32	40	32	40	32	40	32	40	32	40
	16	24	16	24	16	24	16	24	16	24	16	24	16	24	16	24
	48	56	48	56	48	56	48	56	48	56	48	56	48	56	48	56

Table 11: Twiddle exponent index for the first four radix-2 stages.

Stage	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
5	0	4	16	20	8	12	24	28	32	36	48	52	40	44	56	60
	0	4	16	20	8	12	24	28	32	36	48	52	40	44	56	60
	0	4	16	20	8	12	24	28	32	36	48	52	40	44	56	60
	0	4	16	20	8	12	24	28	32	36	48	52	40	44	56	60
6	0	2	8	10	4	6	12	14	16	18	24	26	40	22	28	30
	0	2	8	10	4	6	12	14	16	18	24	26	40	22	28	30
	32	34	40	42	36	38	44	46	48	50	56	58	72	54	60	62
	32	34	40	42	36	38	44	46	48	50	56	58	72	54	60	62
7	0	1	4	5	2	3	6	7	8	9	12	13	10	11	14	15
	32	33	36	37	34	35	38	39	40	41	44	45	42	43	46	47
	16	17	20	21	18	19	22	23	24	25	28	29	26	27	30	31
	48	49	52	53	50	51	54	55	56	57	76	61	58	59	62	63

Table 12: Twiddle exponent index for the last three radix-2 stages.

$q$	$i$	Twid. exp.
0	$0 - -P$	0
1	$\frac{P}{4}$ to $\frac{P}{2} - 1$	0
	$\frac{3P}{4}$ to $P - 1$	$\frac{P}{4}$
2	$\frac{P}{8}$ to $\frac{2P}{8} - 1$	0
	$\frac{3P}{8}$ to $\frac{4P}{8} - 1$	$\frac{2P}{8}$
	$\frac{5P}{8}$ to $\frac{6P}{8} - 1$	$\frac{P}{8}$
	$\frac{7P}{8}$ to $P - 1$	$\frac{3P}{8}$
3	$\frac{P}{16}$ to $\frac{2P}{16} - 1$	0
	$\frac{3P}{16}$ to $\frac{4P}{16} - 1$	$\frac{4P}{16}$
	$\frac{5P}{16}$ to $\frac{6P}{16} - 1$	$\frac{2P}{16}$
	$\frac{7P}{16}$ to $\frac{8P}{16} - 1$	$\frac{16}{16}$
	$\frac{9P}{16}$ to $\frac{10P}{16} - 1$	$\frac{6P}{16}$
	$\frac{11P}{16}$ to $\frac{12P}{16} - 1$	$\frac{1P}{16}$
	$\frac{13P}{16}$ to $\frac{14P}{16} - 1$	$\frac{16}{5P}$
	$\frac{15P}{16}$ to $P - 1$	$\frac{16}{3}$
		$\frac{16}{7P}$
		$\frac{16}{16}$

Table 13: Twiddle exponents for the first four radix-2 stages.

Stage	No of twiddles	Location used	Exponent set
$q$	1	$2^{r-1} - 2^r - 1$	$(\{i_{p-q-1}\})(i_{p-q} \dots i_{p-1} 2^{p-q-1})$
$q + 1$	2	$2^{r-2} - 2 \cdot 2^{r-2} - 1$	$(\{i_{p-q-2} = 1\})(0)i_{p-q} \dots i_{p-1} 2^{p-q-2}$
		$3 \cdot 2^{r-2} - 4 \cdot 2^{r-2} - 1$	$(\{i_{p-q-2} = 1\})(1)i_{p-q} \dots i_{p-1} 2^{p-q-2}$
$q + 2$	4	$2^{r-3} - 2 \cdot 2^{r-3} - 1$	$(\{i_{p-q-3} = 1\})(00)i_{p-q} \dots i_{p-1} 2^{p-q-3}$
		$3 \cdot 2^{r-3} - 4 \cdot 2^{r-3} - 1$	$(\{i_{p-q-3} = 1\})(10)i_{p-q} \dots i_{p-1} 2^{p-q-3}$
		$5 \cdot 2^{r-3} - 6 \cdot 2^{r-3} - 1$	$(\{i_{p-q-3} = 1\})(01)i_{p-q} \dots i_{p-1} 2^{p-q-3}$
		$7 \cdot 2^{r-3} - 8 \cdot 2^{r-3} - 1$	$(\{i_{p-q-3} = 1\})(11)i_{p-q} \dots i_{p-1} 2^{p-q-3}$

Table 14: Local twiddle exponents in multi-sectioning radix-2 DIT FFT. No 90-degree rotations on-the-fly.

Algorithm	Twiddle storage
Pipelined	$n + \frac{P}{2N}$
Bi-section	$n + \frac{P}{2N}$
$2^r$ -section	$\lfloor \frac{n}{r} \rfloor 2^{r-1} + 2^{n-r \lfloor \frac{n}{r} \rfloor - 1} + 2^{p-n-1}$

Table 15: Local twiddle storage requirement in each node with 90-degree rotations on-the-fly.

$$\begin{aligned} \text{No 90-degree rotations: } & \lfloor \frac{n}{r} \rfloor (2^{r-1}) + 2^{n \bmod r-1} + 2^{p-n} - 1 \\ \text{90-degree rotations: } & \lfloor \frac{n}{r} \rfloor 2^{r-2} + \lceil 2^{n \bmod r-2} \rceil + 2^{p-n-1} \end{aligned}$$

**Remark 1:** A bi-section algorithm ( $r = 1$ ) requires  $n + 2^{p-n} - 1$  (no 90-degree rotations), or  $n + 2^{p-n-1}$  (90-degree rotations) twiddles in each node. This is precisely the number of twiddle factors required for the algorithm using no explicit permutations FFT [12].

**Remark 2:** A  $2^{p-n}$ -section algorithm requires approximately a factor of  $\frac{n}{(p-n)} + 1$  more twiddle storage than a bi-section algorithm, or an algorithm without explicit permutations. The reason that additional twiddle storage is required is that when no explicit permutation is performed, then the twiddles for all data in a node are identical for all stages corresponding to bits in the node address field. In a  $2^r$ -sectioning algorithm, these twiddles are split among  $2^r$  nodes. Thus, in the algorithm without explicit permutation,  $2^r$  nodes store at most  $2^r$  distinct twiddles, with one twiddle per node. In a  $2^r$ -section algorithm, each of these nodes may store up to  $2^r$  twiddles (with different nodes storing the same twiddles).

### Twiddle storage for radix- $R$ FFT

The twiddle factors for a high radix DIT FFT are computed similarly to the radix-2 twiddles. For a radix- $2^r$  FFT on  $2^p$  data  $q = \{0, 1, \dots, \lfloor \frac{p}{r} \rfloor - 1\}$ . If  $r$  does not divide  $p$ , then the radix of one stage is  $p - r \lfloor \frac{p}{r} \rfloor$ . The exponents for a typical radix- $2^r$  stage are  $(i_{p-qr-1} i_{p-qr-2} \dots i_{p-(q+1)r}) \cdot (i_{p-qr} i_{p-qr+1} \dots i_{p-2} i_{p-1}) 2^{p-(q+1)r}$ . Thus, with the input in normal order, the twiddles for each radix- $2^r$  stage are in normal order with respect to the indices differing for the inputs of each radix- $2^r$  butterfly stage. The factor  $(i_{p-qr} i_{p-qr+1} \dots i_{p-2} i_{p-1}) 2^{p-(q+1)r}$  in the exponent that is shared for all inputs in each radix- $2^r$  stage, is identical to that of radix-2 stage  $qr$  shifted right  $r$  steps.

For  $r > 1$ , the 90-degree rotation property used to save twiddle factor storage in the radix-2 case can no longer be applied. If there is more than one radix- $2^r$  stage local to a node, then the largest shared rotation factor is  $2^{p-1-r}$ . For  $r > 1$ , complex arithmetic is required to evaluate this factor.

The twiddle storage requirements are summarized in Table 15. It is assumed that 90-degree rotations on-the-fly are performed.

Clearly, the twiddle storage requirement is minimized for the bi-section algorithm, and the algorithms without explicit permutation.

### Selecting the section size and FFT radix.

Algorithm	Comm. requirements
Pipelined	$n \frac{P}{N}$
Bi-section	$(n + 1) \frac{P}{2N}$
$2^m$ -section	$((\lfloor \frac{n}{m} \rfloor + 1) \frac{2^m - 1}{2^m} + \frac{2^{n-m} \lfloor \frac{n}{m} \rfloor - 1}{2^{n-m} \lfloor \frac{n}{m} \rfloor}) \frac{P}{N}$

Table 16: Communication time for a node limited communications system.

The number of bits included in the multi-sectioning and the radix of the FFT can be chosen independently. The number of bits chosen for the multi-sectioning affects the

- communication time
- the range of radices possible for the FFT computation after the multi-sectioning.

The radix of the local FFT after a multi-section stage affects the

- twiddle factor storage requirement
- the efficiency of the arithmetic operations

Let the multi-section be made as a  $2^m$ -sectioning, while the FFT is computed with radix  $2^r$ . The communication times for the FFT are summarized in Table 16 under the assumption that the communication is limited by the amount of data entering or leaving a node.

After a bi-section stage the computations of one radix-2 stage can be performed. After a  $2^m$ -section step, a radix- $2^m$  FFT stage can be performed. This stage can be performed as a combination of lower radix stages, for instance, radix-2, radix-4, and radix-8 stages. Consider a case with 16 data elements per node, and 4096 nodes. Then, a total of 5 (4+1) 8-sections allows the FFT to be computed using 5 stages of radix-8 kernels, and a single radix-2 stage. With 4 (3+1) stages of 16-sections, the computations may be performed as one radix-8 stage plus one radix-2 stage repeated four times, or in total, 4 radix-8 stages and 4 radix-2 stages.

The arithmetic performance is typically maximized when the radix is as large as is compatible with the size of the register file (or cache) associated with the arithmetic unit. The radix of the FFT is limited by the number of bits in the multi-sectioning, i.e.,  $r \leq m$ . The gain in performance for  $r > \text{const } \log_2 W$ , where  $W$  is the number of registers, or the cache size, is typically very limited. For many processors with 32 or 64 registers, choosing  $r = 3$  or  $r = 4$  is optimal. For the Connection Machine system CM-200  $r = 3$  is optimal [12].

## 2.6 Communication requirements

The FFT algorithms we have presented above use the following communication primitives:

- butterfly network emulation
- all-to-all personalized communication

and for the reordering

- bit-reversal
- $k$  steps of shuffle permutations
- a combination of shuffle and bit-reversal

In addition, the Real-to-complex and complex-to-real FFT requires index-reversal which closely resembles vector reversal.

### 3 Multi-dimensional FFT

Computing an FFT along a single axis of a multi-dimensional array implies a number of independent one-dimensional FFTs. The number of FFTs is determined by the product of the length of the axes on which the FFT is not performed. Let the length of the axes be  $P_i$  and the number of nodes over which axis  $i$  is distributed be  $N_i$ . Then, the number of instances of the *problem axis*, i.e., the axis for which an FFT is desired, is  $\prod_{k \neq i} \frac{P_k}{N_k}$ . For the in-place algorithm the data transfer requirements are  $\frac{P_i}{N_i} \log_2 N_i \prod_{k \neq i} \frac{P_k}{N_k}$  in the node limited communication model and  $\prod_k \frac{P_k}{N_k} + n_i - 1$  in the *all-port* communication model. Thus, with  $\prod_k N_k = N$  fixed, in the node limited model it is of interest to minimize  $\log_2 N_i$ , and if possible make the entire axis local to a node. This is also true in the *all-port* communication model.

A multi-dimensional transform is defined by

$$X(k_0, k_1, \dots, k_{m-1}) = \sum_{j_0=0}^{P_0-1} \sum_{j_1=0}^{P_1-1} \dots \sum_{j_{m-1}=0}^{P_{m-1}-1} \omega_{P_0}^{k_0 j_0} \omega_{P_1}^{k_1 j_1} \dots \omega_{P_{m-1}}^{k_{m-1} j_{m-1}} x(j_0, j_1, \dots, j_{m-1}),$$

where  $\omega_{P_i} = \exp -\frac{2\pi i}{P_i}$ . In the node limited model the total communication time is  $\frac{P}{N} \sum_l \log_2 N_l$ , where  $l$  ranges over the axes being transformed. Thus, the product of the axes lengths over which the data set is distributed shall be minimized. In the *all-port* communication model and an in-place algorithm, it is again desirable to minimize the number of nodes over which the axes to be transformed are distributed. However, there is only a very small dependence for large data sets per node.

For the in-place algorithm it is always beneficial with respect to performance to minimize the number of nodes over which the axes to be transformed are distributed. But, the gain is relatively insignificant in the *all-port* communication model.

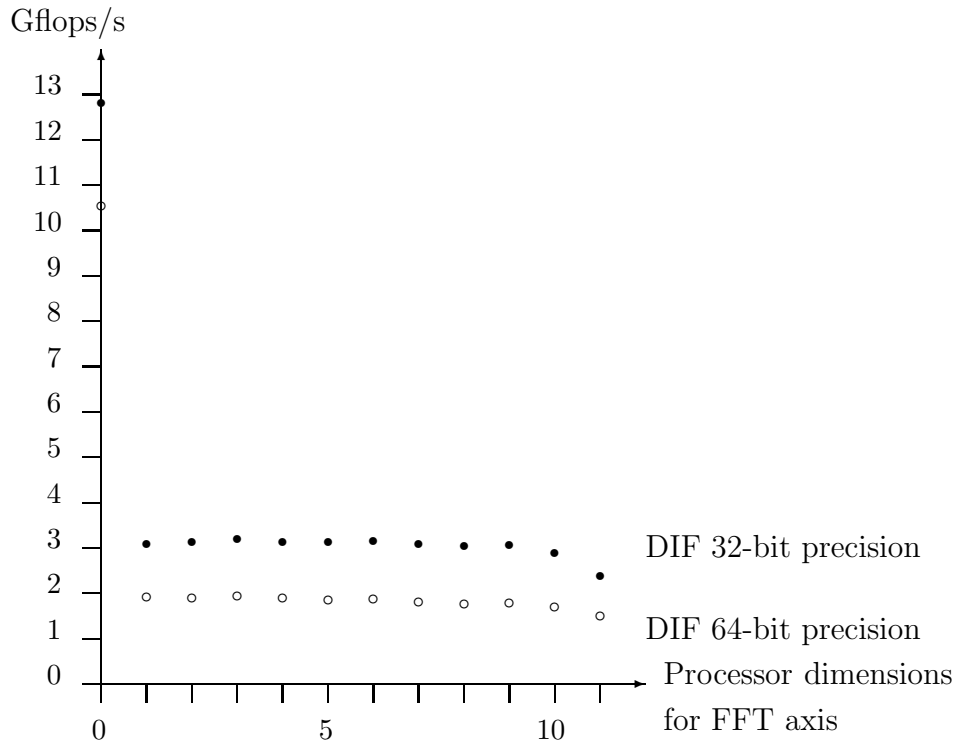


Figure 31: Total execution rate for one-dimensional, unordered, CCFFT on a  $4096 \times 4096$  array as a function of the configuration of a 2048 processor CM-200.

It is easy to verify that the same conclusions hold with respect to the bi- and multi-section algorithms.

For multi-dimensional FFT each axis has its set of twiddle factors. The twiddle factors for an axis is a subset of the twiddle factors for the longest axis. With axes of length  $P_1 \times P_2 \times \dots \times P_k$  the minimum number of twiddle factors is  $\max_{\ell} (R - 1) \frac{P_{\ell}}{R}$ . With separate storage of the twiddle factors for each axis the total storage is  $\sum_{\ell} (R - 1) \frac{P_{\ell}}{R}$ , which is still less than the storage required for a one-dimensional FFT of size  $\prod_{\ell} P_{\ell}$ .

So, in conclusion minimizing the number of nodes over which the axes subject to transformation is distributed is a good idea for all cases we considered. In the *all-port* communication model performance gain is very limited for large data sets per node. For a data set of size  $P$  and axes of equal length, the twiddle factor storage is reduced in proportion to the number of axes, compared to a one-dimensional transform.

The performance of transforms on multi-dimensional arrays is illustrated in Figures 31 – 32. Figure 31 shows that there is a significant advantage to the problem axis being local. Once it is distributed the number of nodes over which it is distributed is of little significance. Figure 32 shows the performance when both axes of the same array are subject to transformation. Making one of the axes local yields the best performance also in this case. But, once both axes are distributed, the performance is again very insensitive to the actual data distribution.

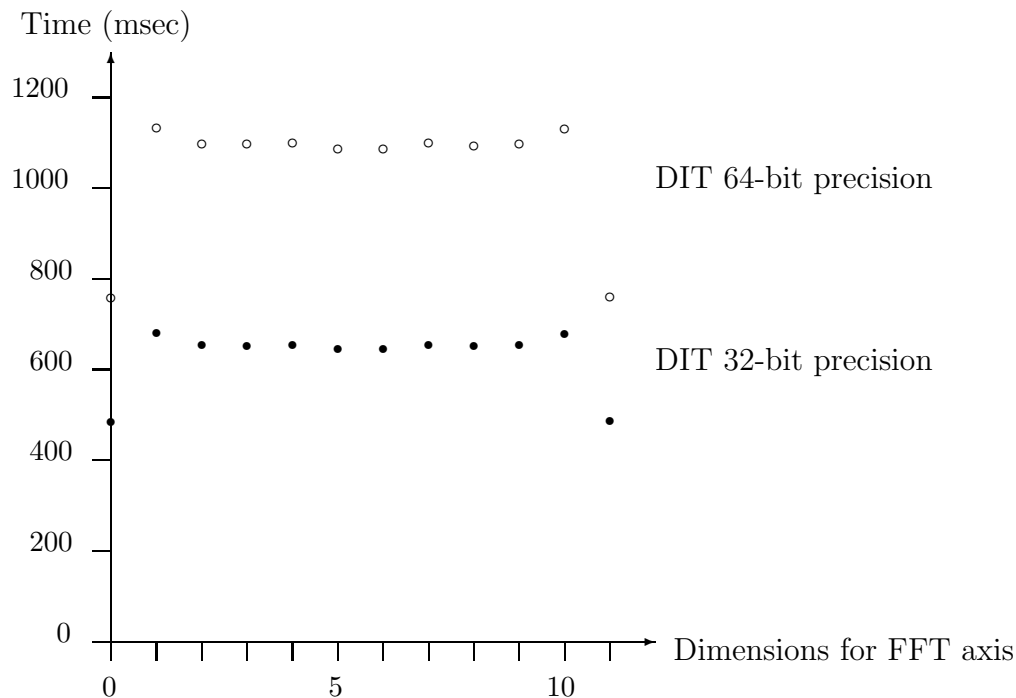


Figure 32: Total execution time for a two-dimensional unordered CCFFT on a  $4096 \times 4096$  array as a function of the configuration of 2048 CM-200 processors.

## References

- [1] James C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp*, 19:291–301, 1965.
- [2] High Performance Fortran Forum. High Performance Fortran; language specification, version 1.0. *Scientific Programming*, 2(1 - 2):1–170, 1993.
- [3] W. Morven Gentleman and G. Sande. Fast Fourier transforms – for fun and profit. In *Proceedings – Fall Joint Computer Conference, 1966*, pages 563–578. AFIPS, 1966.
- [4] Roger W. Hockney. A fast direct solution of Poisson’s equation using Fourier analysis. *J. ACM*, 12:95–113, 1965.
- [5] J.W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 326–333. ACM, 1981.
- [6] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.
- [7] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on Boolean n-cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

- [8] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.
- [9] S. Lennart Johnsson and Ching-Tien Ho. Maximizing channel utilization for all-to-all personalized communication on Boolean cubes. In *The Sixth Distributed Memory Computing Conference*, pages 299–304. IEEE Computer Society Press, 1991.
- [10] S. Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, volume 826, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.
- [11] S. Lennart Johnsson, Michel Jacquemin, and Robert L. Krawitz. Communication efficient multi-processor FFT. *Journal of Computational Physics*, 102(2):381–397, October 1992.
- [12] S. Lennart Johnsson and Robert L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing*, 18(11):1201–1221, 1992.
- [13] S. Lennart Johnsson, Robert L. Krawitz, Douglas MacDonald, and Roger Frye. A radix-2 FFT on the Connection Machine. In *Supercomputing 89*, pages 809–819. ACM, November 1989.
- [14] S. Lennart Johnsson, Uri Weiser, Danny Cohen, and Al Davis. Towards a formal treatment of VLSI arrays. In *Proceedings of the Second Caltech Conference on VLSI*, pages 375 – 398. Caltech Computer Science Dept., January 1981.
- [15] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, 1982.
- [16] Alan V. Oppenheimer and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs. NJ, 1975.
- [17] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.
- [18] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.
- [19] Charles van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.